ВЕЛИКОТЪРНОВСКИ УНИВЕРСИТЕТ
"СВ. СВ. КИРИЛ И МЕТОДИЙ"
Факултет "Математика и информатика"
Катедра "Информационни технологии"

Дисертационен труд
на **Душан Илия Биков**
на тема:

Криптографски свойства на някои векторни булеви функции и паралелни алгоритми с CUDA

за присъждане на образователна и научната
степен "доктор"
Професионално направление
4.6. "Информатика и компютърни науки"

научен ръководител:
проф. дмн Стефка Христова Буюклиева

Велико Търново
2017 г.

С Ъ Д Ъ Р Ж А Н И Е

Списък на съкращенията

AC - Autocorrelation
ACT - Autocorrelation Transform
ALU - Arithmetic Logic Unit
ANF - Algebraic Normal Form
BoolSPLG - Boolean S-box Properties Library for GPUs
CPU - Central Processing Unit
CUDA - Compute Unified Device Architecture
DDT - Difference Distribution Table
FFT - Fast Fourier Transform
FLOP - Floating-point Operations per second
FMT - Fast Möbius Transform
FPU - Floating Point Unit
FWT - Fast Walsh Transform
GPU - Graphics Processing Unit
IFWT - Inverse Fast Walsh Transform
LAT - Linear Approximation Table
PTT - Polarity Truth Table
QC - Quasi-Cyclic
QCS-boxes - Quasi-Cyclic S-boxes
RM - Reed-Muller
SFU - Special Function Unit
SIMD - Single Instruction Multiple Data
SIMT - Single Instruction, Multiple Threads
SMs - Streaming Multiprocessor
SP - Streaming Processor
SPMD - Single Program Multiple Data
TT - Truth Table

Увод

Гарантирането на сигурността, поверителността, целостта и достъпността на информацията днес е от съществено значение. Основна роля за това имат блоковите шифри и хеш-функции, които днес широко се използват за осъществяване на сигурна комуникация. Модерните блокови шифри (и хеш-функции) имат един или повече нелинейни слоеве, които осигуряват ефект на объркване (confusion) [60], което е от жизнена важност за сигурността. Векторна булева функция (наричана още (n, m) S-box, накратко S-box или субституционна кутия) е криптографски примитив, който се използва за оформяне на нелинейния слой на блоковите шифри. Има добре изследвани критерии, които дадена векторна булева функция трябва да удовлетворява, за да бъде съответният шифър устойчив на различни видове атаки.

За представяне, определяне и пресмятане на характеристиките на векторните булеви функции са необходими ефективни алгоритми. С нарастването на размера на векторната булева функция (броя на променливите) свързаните задачи стават по-непосилни за изчисление. Затова важна перспектива дава фактът, че алгоритмите за тяхното решаване са подходящи за паралелна реализация. Реализацията на паралелни алгоритми и паралелни изчисления през последните години е възможна и със съвременните персонални компютри, тъй като те имат няколко процесора, процесори с повече ядра или имат графични платки, които включват процесор или процесори с повече ядра.

В това изследване представяме паралелна реализация на библиотека за изследване на важни криптографски свойства и пресмятане на параметри на булевите и векторните булеви функции. Намирането на векторни булеви функции с добри криптографски свойства е трудна задача, особено за размер $n \geq 8$. В тази дисертация ще представим метод за конструиране на векторни булеви функции с добри криптографски свойства чрез използване на квазициклични кодове.

Булеви функции.

Булевите функции са едни от основните обекти в дискретната математика и свързаните с нея дисциплини. *Булева функция* на n променливи f е изобра-

жение от $\mathbb{F}_2^n$ в $\mathbb{F}_2$, където $\mathbb{F}_2 = \{0, 1\}$ е полето с 2 елемента, а $\mathbb{F}_2^n$ е n-мерното векторно пространство над това поле.

Две от най-естествените представяния на булева функция f на n променливи са чрез таблица на истинност и алгебрична нормална форма. *Таблицата на истинност* $TT(f)$ (Truth Table) е двоичен вектор с дължина 2^n, чиито координати са стойностите на функцията f, като i-тата координата е равна на $f(\bar{i})$, където $\bar{i}$ е двоичният запис на цялото число i (разгледан като двоична n-орка), $i = 0, 1, \ldots, 2^n - 1$.

С всяка булева функция f се асоциира функцията $(-1)^f = 1 - 2f$, чиито стойности са от множеството $\{-1, 1\}$. Множеството от стойности на функцията $(-1)^f$ се записват във вектор, който се нарича *Polarity Truth Table* (PTT) и се бележи с $PTT(f)$, $PTT(f)[i] = (-1)^{TT[i]}$, $i = 0, 1, \ldots, 2^n - 1$.

Тегло (по Хеминг) $\mathrm{wt}(x)$ на вектора $x = (x_1, x_2, ..., x_n) \in \mathbb{F}_q^n$ наричаме броя на ненулевите му координати:

$$\mathrm{wt}(x) = |\{i | x_i \neq 0\}|.$$

Под тегло $\mathrm{wt}(f)$ на функцията f ще разбираме теглото на вектора $TT(f)$.

Разстояние (по Хеминг) $d(x, y)$ между два вектора $x = (x_1, x_2, ..., x_n)$ и $y = (y_1, y_2, ..., y_n)$ от $\mathbb{F}_q^n$ наричаме броя на координатите, в които те се различават:

$$d(x, y) = |\{i | x_i \neq y_i\}|.$$

Аналогично под разстояние между две функции ще разбираме разстоянието между съответните таблици на истинност.

Друго естествено представяне на булевата функция f е като двоичен полином на n променливи, в който всеки едночлен е произведение на променливи от нулева или първа степен:

$$f(x_1, x_2, \ldots, x_n) = \bigoplus_{u=0}^{2^n-1} a_u x^u,$$

където $a_u \in \mathbb{F}_2$, $x^u = x_1^{u_1} x_2^{u_2} \cdots x_n^{u_n}$, $u = (u_1, \ldots, u_n) \in \mathbb{F}_2^n$. Това представяне е също еднозначно определено и е известно като *алгебрична нормална форма* (Algebraic Normal Form, ANF, полином на Жегалкин)[11]. Степента на този полином се нарича *алгебрична степен (algebraic degree)* на f и се бележи с $deg(f)$.

Walsh коефициентите $f^W(a)$ се дефинират чрез трансформацията на Walsh (Walsh-Hadamard transform):

$$f^W : \mathbb{F}_2^n \to \mathbb{Z}, \quad f^W(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus f_a(x)} = 2^n - 2d_H(f, f_a),$$

където $a = (a_1, \ldots, a_n) \in \mathbb{F}_2^n$, $f_a(x) = a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n$ и $f(x_1, x_2, \ldots, x_n) \oplus a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n = f(x) \oplus f_a(x)$. При лексикографска наредба на векторите от $\mathbb{F}_2^n$, можем да наредим Walsh коефициентите $f^W(a)$ и да ги разглеждаме като координати на вектор. Този вектор наричаме Walsh спектър на булевата функция f [11].

Линейност $Lin(f)$ на булевата функция f е най-голямото число измежду абсолютните стойности на коефициентите на Walsh: $Lin(f) = \max\{|f^W(a)| \mid a \in \mathbb{F}_2^n\}$. За линейността е в сила следното неравенство: $Lin(f) \geq 2^{n/2}$ [11]. Вместо линейност в много изследвания се търси *нелинейност* $nl(f)$, която се дефинира като минималното разстояние от функцията f до афинните булеви функции, или

$$nl(f) = \min\{d(f, a_0 \oplus a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n),\ a_i \in \mathbb{F}_2,\ i = 0, 1, \ldots, n\}.$$

Връзката между линейност и нелинейност на булева функция се дава с формулата $2nl(f) + Lin(f) = 2^n$.

Автокорелация AC на булевата функция f се дефинира чрез автокорелационната трансформация:

$$r_f : \mathbb{F}_2^n \to \mathbb{Z}, \quad r_f(w) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus f(x \oplus w)},\ w \in \mathbb{F}_2^n.$$

Автокорелация на булева функция е най-голямото число измежду абсолютните стойности на автокорелационните коефициенти: $AC(f) = \max\{|r_f(w)| \mid w \in \mathbb{F}_2^n\}$.

Векторни булеви функции.

Векторна булева функция (наричана още (n, m) S-box, кратко S-box или субституциона кутия) може да се разглежда като функция S, която на редица от n входни бита съпоставя редица от m изходни бита (в много случаи се задава с таблица):

$$S : \mathbb{F}_2^n \to \mathbb{F}_2^m.$$

Векторна булева функция може да се представи като вектор $(f_1, f_2, \ldots, f_m)$, където f_i са булеви функции на n променливи, $i = 1, 2, \ldots, m$. Функциите f_i се наричат *координатни функции* на векторната булева функция. Те могат да се представят чрез техните таблици на истинност $TT(f_i)$ и по този начин векторната булева функция S може да се разглежда като матрица:

$$G(S) = \begin{pmatrix} TT(f_1) \\ TT(f_2) \\ \ldots \\ TT(f_m) \end{pmatrix}.$$

Една векторна булева функция е обратима (биективна) тогава и само тогава, когато $m = n$ и матрицата $G(S)$ поражда $[2^n, n, 2^{n-1}]$ симплекс код S_n с добавен нулев стълб в началото.

За изучаване на някои от криптографските свойства (линейност, нелинейност, алгебрична степен) на векторните булеви функции, трябва да се изследват всички ненулеви линейни комбинации на координатните функции, означени с

$$S_b = b \cdot S = b_1 f_1 \oplus \cdots \oplus b_m f_m,$$

където $b = (b_1, ..., b_m) \in \mathbb{F}_2^m$. Тези линейни комбинации се наричат *компонентни булеви функции*. Walsh спектър на векторните булеви функции се дефинира като множество от всички Walsh спектри на компонентните булеви функции.

Линейност и нелинейност на векторна булева функция S се дефинира като:

$$Lin(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} Lin(b \cdot S), \qquad nl(S) = \min_{b \in \mathbb{F}_2^m \setminus \{0\}} nl(b \cdot S).$$

Автокорелация AC на векторна булева функция S дефинираме като:

$$AC(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} AC(b \cdot S).$$

Диференциална еднаквост (Differential uniformity) δ е друг важен криптографски параметър на векторните булеви функции. Дефинира се със следната формула:

$$\delta = \max_{\alpha \in \mathbb{F}_2^n \setminus \{0\}, \beta \in \mathbb{F}_2^m} |\{x \in \mathbb{F}_2^n | S(x) \oplus S(x \oplus \alpha) = \beta\}|.$$

Стойността на δ трябва да е колкото е възможно по-ниска. За параметъра δ са в сила следните граници $2^{n-m} \leq \delta \leq 2^n$. За обратими векторни булеви функции $\delta \geq 2$.

Квазициклични кодове.

Нека $\mathbb{F}_q$ е крайно поле с q елемента, а $\mathbb{F}_q^n$ е n-мерното векторно пространство над $\mathbb{F}_q$. Всяко k-мерно подпространство C на $\mathbb{F}_q^n$ наричаме *линеен код* с дължина n и размерност k над $\mathbb{F}_q$. Кодовете над полето $\mathbb{F}_2$ ще наричаме *двоични кодове*. Ще разглеждаме само двоични линейни кодове.

Даден линеен код е квазицикличен, ако всяко циклично отместване на кодова дума с $s \geq 1$ позиции дава като резултат друга кодова дума. Ако $s = 1$ кодът е цикличен, следователно квазицикличните (QC) кодове са обобщение на цикличните кодове.

Нека $K = \mathbb{F}_{2^n}$ е крайно поле, α е негов примитивен елемент, $2^n - 1 = m \cdot r$, и $\beta = \alpha^r$. Ако $G = \langle \beta \rangle < K^*$, то G е циклична група от ред m, а $G, \alpha G, \alpha^2 G, \ldots, \alpha^{r-1} G$ са всички различни съседни класове на G в K^*. За $a \in \mathbb{Z}_r$ дефинираме циркулантна $m \times m$ матрица $C_a = (Tr(\alpha^{ma} \beta^{i+j}))_{0 \leq i,j \leq m-1}$. Когато m и r се взаимно прости, матриците C_a съответстват на различните съседни класове на G в K^*.

Ако m и r са взаимно прости, кодът $C(M)$, чиито ненулеви кодови думи са редовете на матрицата

$$M = \begin{pmatrix} C_0 & C_1 & \ldots & C_{r-1} \\ C_{r-1} & C_0 & \ldots & C_{r-2} \\ & & \vdots & \\ C_1 & C_2 & \ldots & C_0 \end{pmatrix}, \tag{1}$$

е еквивалентен на симплекс $[2^n - 1 = mr, n, 2^{n-1}]$ кода S_n.

Нека $\overline{M}$ е разширена матрица на M с добавен нулев стълб в началото, а $C(\overline{M})$ е код, чиито кодови думи са редове на $\overline{M}$. Тогава всяка пораждаща матрица на $C(\overline{M})$ може да се разглежда като обратима векторна булева функция.

Кратка история на задачата за класификация и конструиране на векторни булеви функции с добри криптографски свойства.

Класификацията и конструирането на векторните булеви функции с добри криптографски свойства е трудна задача, особено за размер $n \geq 8$. Криптографските свойства на векторните булеви функции, както и различни конструкции, се изследват в продължение на много години. Техниките за конструиране на векторни булеви функции включват: алгебрични структури, псевдослучайно генериране и различни евристични подходи.

Обратимите векторни булеви функции от размер $n = 4$ са широко и подробно изследвани, направена е класификация [36, 57, 64], дефинирана е оптималност [36]. Обща класификация на всички оптимални векторни булеви функции за размер $n = 4$ е дадена в работата на Leander и Poschmann [36] през 2007 г. Авторите правят изчерпателен анализ и намират всичките класове на афинна еквивалентност, за които изследват линейност, диференциална еднаквост и алгебрична степен. Saarinen [57] през 2011 г., разширява работата на Leander и Poschmann, при което прави изчерпателен анализ и намира всичките класове на линейна еквивалентност. В статията от 2015 г. [64] се прави нова класификация на векторните булеви функции с размер $n = 4$, при която те се разделят на 183 различни категории, получени в зависимост от стойностите на подходящо дефинирани от авторите параметри.

За разлика от векторните булеви функции с размер $n = 4$, изследванията и направените класификаии при по-големи размери са все още много непълни. Специален интерес има към векторните булеви функции с размер $n = 8$, тъй като обратими векторни булева функции с този размер са внедрени в някои от най-разпространените криптографски стандарти. Още не е ясно дали съществуват векторни булеви функции за $n = 8$ с нелинейност, по-голяма от 112 (каквато е нелинейността на векторната булева функция, използвана в блоковия шифър AES). Според неравенството на Парсевал нелинейността в този случай е ≤ 120, но нито е доказано, че стойностите между 113 и 120 не са възможни, нито е конструирана векторна булева функция за $n = 8$ с по-голяма от 112 нелинейност. Поради това намирането на векторни булеви функции с по-добри "оптимални" криптографски свойства за $n = 8$ е много актуална задача. Интересни са и конструкциите на векторни булеви функции с по-големи размери.

Паралелно програмиране.

Паралелна изчислителна платформа в общия случай представлява компютърна система с множество от изчислителни единици (процесори), която поддържа паралелно програмиране. Паралелното програмиране е програмиране на език, който позволява изрично посочване на различни порции от изчисления да се изпълняват едновременно от различни процесори (ядра). Всъщност паралелното изчисление представлява модел на изчисление, при което множества от изчисления се изпълняват едновременно (еднакви малки програми правят изчисления над различни данни), действайки по принципа, че големи задачи често могат да се разделят на по-малки, които след това да бъдат решени едновременно (паралелно).

Компютърната система, която използваме за реализацията на алгоритмите, има компютърна архитектура SIMD (Single Instruction Multiple Data), според класификацията на Flynn [21, 22, 23]. SIMD архитектурите се състоят от десетки независими процесори, които могат едновременно да изпълняват един поток от команди над различни потоци от данни. Приема се, че всеки процесор изпълнява еднакви програми. Тази компютърна архитектура се използва в случаите, когато много изчисления са необходими с всички процесори върху една и съща работа. Използваният паралелен модел на изпълнение влиза в категорията SPMD (Single Program Multiple Data). При SPMD множество независими процесори изпълняват едновременно една и съща програма над различни данни.

За нашите изчисления ние използваме персонален компютър, който има подходяща графична платка, поддържаща паралелно програмиране. Напосле-

дък има ускорено развитие на паралелната изчислителна платформа и програмен модел CUDA (Compute Unified Device Architecture). Програмите от настоящия труд са написани предимно с CUDA C, което е развойна среда с общо предназначение, с основа на C/C++ стандарта и с набор от разширения, които позволяват хетерогенно програмиране.

В зависимост от архитектурата NVIDIA графичните платки са организирани в SMs (Streaming Multiprocessors), контролер на паметта, както и различна йерархия на паметта, която съдържа: набор от регистри, константи, текстура кеш (texture caches), споделена памет (shared memory) и глобална памет (global memory). Всеки SM съдържа няколко SPs (Streaming Processors - общоприето е наименованието CUDA ядро). SM е изграден да изпълнява хиляди нишки едновременно. За да управлява такава голяма бройка от нишки, използва уникална компютърна архитектура, наречена SIMT (Single-Instruction, Multiple-Thread) [16], която е сходна с SIMD архитектурата. При архитектурата SIMT е позволено програмистите да пишат паралелен код на ниво на нишка, което позволява независимост, скалируемост, както и координиране на нишките.

Ще отбележим, че CUDA C осигурява възможности за оптимизация на паралелните алгоритми. Почти винаги можем да оптимизираме CUDA C кода, което ни осигурява по-ефективно изпълнение на паралелната програма.

В настоящата работа реализираме паралелна библиотека за изчисление на някои свойства на булевите и векторните булеви функции. Освен това представяме метод за конструиране на добри векторни булеви функции, а процедурите от библиотеката използваме за реализиране на алгоритмите, свързани с този метод.

Изследванията ни са структурирани в следните глави:

В **Глава 1** са представени основни понятия както за булевите и векторните булеви функции, така и за паралелното програмиране. Освен това е представена паралелна изчислителна платформа и програмен модел CUDA.

Начални изследвания, свързани с криптографските свойства, характеристики и класификация на векторни булеви функции (S-boxes), са представени в [P1]. Относно паралелното програмиране първоначално се спряхме на умножението на вектор по матрица, чиято паралелна реализация е тривиална. В публикацията [P3], която е докладвана на конференция [D3], е представена паралелна реализация на умножение на вектор по матрица, свързана със спектъра на Walsh, на което по-подробно ще се спрем в следващите глави.

В **Глава 2** се обсъждат някои трансформации на булеви функции, чието описание се прави много естествено, чрез двоичното представяне на целите

неотрицателни числа. Основните точки на тази глава са представяне на някои алгоритми с приложение в криптографията. На практика разглежданите алгоритми се реализират чрез умножение на вектор с матрица.

Булевите функции са едни от основните обекти в дискретната математика и свързаните с нея дисциплини. Тук ние ги разглеждаме от гледна точка на криптографските им свойства. Една булева функция може да се опише по много различни начини. Две от най-естествените представяния на булева функция на n променливи са чрез таблица на истинност и алгебрична нормална форма. Представянето на булевата функция чрез таблица на истинност е много удобно за описание на някои трансформации и представяния на булеви функции и на алгоритми, свързани с тях. Методът за пресмятане на таблицата на истинност при зададена алгебрична нормална форма и обратно, който използваме, се нарича бърза трансформация на Мьобиус (Fast Möbius Transform). В литературата често срещано наименование на този алгоритъм е трансформация на Reed–Muller [31].

Използване на двоичното представяне на целите неотрицателни числа представлява основната идея за реализация на нашия алгоритъм. От криптографска гледна точка много важен параметър за една булева функция f е нейната нелинейност, която е свързана с разстоянието от f до афинните функции и намирането на спектъра на *Walsh*[11]. Другата ни цел е представяне на ефективен алгоритъм за пресмятане на спектъра на *Walsh* и оттам и на линейността и нелинейността на дадена булева функция. За пресмятането на Walsh спектъра ние използваме така наречената бърза трансформация на Walsh, която е подобна на бързата трансформация на Мьобиус. Освен това двата алгоритъма имат еднаква сложност $O(n2^n)$. И двата алгоритъма изцяло се определят от двоичното представяне на целите числа от 0 до $2^n - 1$.

Важен за нашите алгоритми е фактът, че матриците имат не само рекурсивна структура, но тя е от специален вид, който позволява много ефективно бътерфлай (butterfly) умножение. Въпреки, че предложените алгоритми не са по-добри в порядък от досега известните, те са много по-компактни и много по-нагледни.

Представеният подход е съвместна разработка с Илия Буюклиев и основно се базира на приложението на двоичното представяне на целите неотрицателни числа, за описване на трансформациите и представянето на булевите функции и алгоритми свързани с тях. Алгоритъмът за преобразуване на булеви функции (от таблица на истинност в ANF) може да представлява част от алгоритъм за търсене на алгебрична степен, която е важна криптографска характеристика. Представеният алгоритъм за изчисление на спектъра на Walsh представлява част от последователната версия на алгоритъма за търсене

на добри векторни булеви функции, конструирани от квазициклични кодове, за което ще говорим в петата глава. Основно алгоритмите са публикувани на конференцията на Съюза на математиците в България, проведена през 2015 г. [P2].

В **Глава 3** е представена паралелна реализация на алгоритъм за пресмятане на спектъра на Walsh (FWT) и алгебричната нормална форма (FMT) на булева функция. Тези алгоритми реализираме в CUDA и правим сравнение с последователно реализираните алгоритми. Както при последователните алгоритми (Алгоритъм 2.1 и Алгоритъм 2.2), така и тук използваме двоично представяне на целите неотрицателни числа. В тази глава посочваме и ефективността на паралелното изчисление, като можем да видим ползите от неговото използване.

Представили сме няколко паралелни алгоритъма за пресмятане на спектъра на Walsh, при които използваме основните концепции на Алгоритъм 2.2. Нашата мотивация за създаване на различни алгоритми дължим на факта, че има различни стратегии за оптимизация и техники със специфични характеристики. Първият алгоритъм е основен и по-лесен за имплементация в CUDA. За увеличаване на производителността правим по-сложни алгоритми с прилагане на различни оптимизационни техники. Целта тук е да презентираме ефективни алгоритми, да сравним оптимизационните техники и стратегии. Показваме стъпка по стъпка подобряване и оптимизация на основния алгоритъм, което увеличава производителността. Изборът на правилна оптимизационна техника и подходящи методи осигурява това увеличение. При паралелната реализация на FMT, имплементираме вече представените оптимизационни техники.

Важно е да споменем, че при алгоритмите, които използват шаблон за пренареждане на паметта (това важи за FWT при $11 \leq n \leq 20$), стъпката с подреждането на изчислените Walsh коефициенти може да се пропусне. Вместо нея, нишките записват изчислените стойностите директно в глобалната памет според индекса. Самото пропускане на стъпката за пренареждането на изчислените коефициенти спестява ресурс и ускорява изпълнението с 10-20%.

Разработените алгоритми в тази глава са съвместна работа с Илия Буюклиев. Те са описани в статии, които са приети за публикуване или се редактират [P8], [P7]. Някои идеи и предварителни версии на алгоритмите са докладвани на [D2], [D4], [D5]. Също така някои версии на алгоритъма за изчисление на спектъра на Walsh, са публикувани в [P4], [P5].

В раздел 3.3 е представен изцяло различен подход за реализация на бързата трансформация на Walsh. Алгоритъмът в този раздел наричаме компактен алгоритъм, което идва от самата реализация, която разглеждаме като пара-

лелно събиране на два вектора. Тук също така използваме споделена памет, локални регистри и шаблон за пренареждане на паметта. Този подход на реализация при първоначалните изследвания дава резултати, близки до най-добрия алгоритъм, представен в подраздел 3.1.1 (Алгоритъм 3.6), но още не е направен подробен анализ и сравнение. Идеята на този алгоритъм е съвместна работа с Илия Буюклиев, но до момента още не е публикуван.

В **Глава 4** представяме паралелна реализация на библиотека за изчисление на някои свойства на булевите и векторните булеви функции, Boolean S-box Properties Library for GPUs (BoolSPLG). Библиотеката се основава на методология, която използва серия от изграждащи функции, позволяващи конструкция на алгоритми с малко усилия. При реализацията на библиотеката се обръща специално внимание на гъвкавостта и адаптивността. Използва се обобщен подход за проектиране на паралелни алгоритми, при което се получава оптимална производителност.

Алгоритмите за трансформация (Fourier-related transforms) имат приложение в много области (свързани с дискретната математика) като криптографията, теорията на кодирането, компресия на данните и т.н. Задачите, свързани с представянето, дефинирането и пресмятането на най-важните криптографски свойства и параметри на булевите и векторните булеви функции, изискват ефективни алгоритми. С увеличаване на размера на входните данни, за пресмятане се изисква по-голям изчислителен ресурс. За изчисление на някои от криптографските характеристики (линейност, автокорелация, алгебрична степен, диференциална еднаквост) е необходима реализация на ефективни бътерфлай алгоритми.

Нашите изследвания се отнасят основно до конструиране на векторни булеви функции (S-boxes) с добри криптографски свойства [P6]. Броят на получените векторни булеви функции е огромен и затова избираме само тези с добри криптографски свойства. По тази причина имаме нужда от бързи алгоритми за изчисления. За нуждите на нашите изследвания ние разработваме библиотека, която съдържа паралелни алгоритми, написани на CUDA C за GPU [P9].

CUDA ориентираната библиотека ни позволява да изследваме и изчисляваме криптографските свойства и параметри на големи булеви и векторни булеви функции. Библиотеката съдържа набор от различни процедури (паралелни функции), които могат да се ползват за конструкция на други алгоритми, без писане на комплексен код, което спестява време. Предназначението на библиотеката е да улесни и ускори писането на приложения, които конструират векторни булеви функции.

Изграждащите блокове на алгоритмите са организирани в няколко слоя, така че всяка процедура изпълнява определена работа. Възможно е по-ефективно генериране на изпълним файл, като е предоставена допълнителна информация на CUDA *nvcc* компилатора, като размера на входните данни, конфигурация на мрежата от нишки и т.н. Нашите процедури автоматично настройват мрежата от нишки и блокове. Повечето от функциите са проектирани да правят изчисления на ниво регистри поради най-бързата честотна лента. Имплементираните паралелни функции комбинират използваемата памет (регистри, локална памет, споделена памет) и по този начин осигуряват по-висока пропускателна способност.

Предложената библиотека реализира процедури с комбиниране на основните паралелни функции в параметризирана процедура, която освобождава от грижата за подробностите на имплементацията. Изграждащите функции са класифицирани в изчисления на данни (Butterfly (FWT, IFWT, FMT), DDT, AlgebraicDegree, ComponentFunction, PowerInt), манипулация на данни (Copy, MemoryPatern) и операция за подкрепа (reduction). Нашата библиотека съдържа следните бътерфлай алгоритми: binary Fast Walsh Transforms (FWT), binary Inverse Fast Walsh Transforms (IFWT), binary Fast Möbius Transforms (FMT). Освен това има и допълнителни алгоритми (алгоритми за изчисление на DDT, алгебрична нормална форма, компонентните функции) помощни алгоритми и функции (reduction) за осъществяване на необходимите операции.

Представените алгоритми и процедури в библиотеката BoolSPLG са съвместна работа с Илия Буюклиев. Някои идеи и предварителни версии на част от алгоритмите са докладвани на [D2], [D4], [D5]. Също така някои идеи и версии на алгоритъма за изчисление на спектъра на Walsh, са публикувани в [P4], [P5]. Статия за паралелната библиотека BoolSPLG се подготвя за публикуване [P9].

В **Глава 5** представяме метод за конструкция на обратими криптографски векторни булеви функции чрез квазициклични кодове (QC). Получените резултати са дадени в края на главата. Освен това показваме къде и при кои случаи може да приложим паралелни алгоритми в метода за конструиране и търсене на биективни криптографски векторни булеви функции.

Намирането на добри векторни булеви функции е обсъждано в много статии и научни разработки в продължение на десетки години, при което са провеждани много разнообразни експерименти над тях. Векторната булева функция представлява ключов криптографски примитив при дизайна на блоковите шифри. Те формират нелинейния слой на блоковите шифри и поради това са много важни за тяхната сигурност. Всъщност векторните булеви функции се

използват за субституция (замяна), което в процеса на шифриране осигурява разбъркване ($confusion$) [60] - един от основните принципи за практичен дизайн на криптографски алгоритми. Има добре изследвани критерии, които използваната векторна булева функция трябва да удовлетворява, за да бъде шифърът устойчив на различни криптографски атаки, като диференциален и линеен криптоанализ. Въпреки това, конструкцията на криптографски сигурна векторна булева функция още представлява проблем. В продължение на много години са изследвани свойства, различните техники и методи за конструиране на добри криптографски векторни булеви функции. Техниките за конструиране на векторни булеви функции могат да се класифицират в три категории: алгебрични структури, псевдослучайно генериране и различни евристични подходи. Класификацията и намирането на векторна булева функция с добри криптографски свойства е трудна задача, особено за размер $n \geq 8$. Обратимите векторни булеви функции от размер $n = 4$ са широко и подробно изследвани, направена е класификация [36, 57, 64], дефинирано е какво представлява оптимална векторна булева функция [36].

Поради $(2^n)!$ възможни векторни булеви функции, трудно е намирането на оптимални векторни булеви функции за $n \geq 5$ с изчерпващо търсене. Поради тази причина правим структура, която е близо до изчерпващо търсене. Използваме матричното представяне на векторната булева функция, и разделяме стълбовете в групи, над които правим частично търсене. Също така използваме връзката между векторната булева функция и симплекс кода и използваме пораждаща матрица на симплекс кода с циркулантна структура. В нашата конструкция симплекс кодът, е представен като квазицикличен код.

За конструиране на векторни булеви функции използваме двоични QC кодове. Такива обратими векторни булеви функции наричаме QCS-boxes. Използваните двоични QC кодове са конструирани по два метода.

В нашата работа изследваме QCS-boxes за размери $4 \leq n \leq 20$, и оценяваме предложените конструкции. Направихме изчерпващо търсене за всички размери при $r = 15$, а няколко случая още се изследват за $r \geq 17$. За първата конструкция няма да представим експериментални резултати, поради това че получените QCS-boxes имат слаби криптографски свойства. Някои от получените добри QCS-boxes (с различни n) чрез новите методи за конструиране са дадени в шестнадесетичен (hex) запис в приложението, докато основните криптографски параметри са представени в таблична форма в следващите подраздели на тази глава. Тук представяме всичките получени QCS-boxes с определени размери, които се доближават до границата на Парсевал (Теорема 1.6 (Таблица 5.2)), имат най-малката възможна стойност на δ, имат максимална алгебрична степен и минимална възможна стойност $AC(S)$, по което

сравняваме с най-добрите известни векторни булеви функции.

Векторните булеви функции, които се получават чрез първата конструкция, нямат добра нелинейност. Тази конструкция е естествена, но втората и третата конструкция са по-важни, защото дават по-добри резултати. При втората конструкция (за двата метода на генериране на QS кодове) и всички размери на n, когато увеличаваме стойността на параметъра r (m намалява), нелинейността на получените QCS-boxes има тенденция да се приближава до границата на Парсевал (Теорема 1.6). Третата конструкция е разширение на втората, като тук имаме добавено пренареждане на стълбовете за всяка циркуланта отделно. Параметрите на най-добрите получени QCS-boxes са представени таблично, а примери на конкретни QCS-boxes са дадени в приложение.

Както вече описахме, в предните Глави (3 и 4) е реализирана паралелна библиотека BoolSPLG [P9], която реализира алгоритми за изчисление на някои криптографски свойства на векторфни булеви функции. Процедурите в тази библиотека използваме за дизайн на алгоритми за представените конструкции. Полза от прилагането на паралелни алгоритми имаме при големи стойности на n ($n \geq 14$, подраздел 5.5.1). Тест платформите, които използваме за експериментите, са дадени в Таблица 3.1.

Представените конструкции са съвместна разработка с Илия Буюклиев и Стефка Буюклиева. Първите две конструкции и част от резултатите са докладвани на международната конференция $15th\ International\ Workshop\ on\ ACCT$, проведена в Албена, България, през 2016 година [D7] и публикувани в статия [P6]. Подробно описание на всичките конструкции и получени резултати се подготвят за статия, която е съвместна с Илия Буюклиев и Стефка Буюклиева. Представеният паралелен алгоритъм е съвместна работа с Илия Буюклиев.

Бих искал да изразя голямата си признателност към моя научен ръководител проф. Стефка Буюклиева за безценните съвети и огромното търпение при разработването на настоящата дисертация. Искам също да изразя голяма признателност към проф. Илия Буюклиев за съдействието, съвместната работа и безценните съвети, без които представените тук изследвания нямаше да се осъществят. Благодаря на всички, които ми помагаха при подготовката на този труд с препоръки и критични забележки. Изказвам благодарност и на колегите си от катедра "Информационни технологии" към факултет "Математика и информатика" на Великотърновския университет. Също така благодаря на колегите от секция МОИ на ИМИ–БАН и на всички участници в Ежегодния национален семинар по теория на кодирането "Стефан Додунеков".

Изследванатa в тази дисертация са частично финансирани от научно-изследователските проекти:

- Разработка на математически методи за проектиране, оценяване и имплементиране на криптографски схеми за защита на информацията, ФНИ МОН, договор No. ДФНИ-И01/0003 от 2012 г.

- Алгебрични и геометрични модели в кодирането и криптографията и реализирането им чрез паралелни алгоритми, ВТУ, договор No. РД-09-422-13 от 09.04.2014 г.

- Кодове и комбинаторни конфигурации, ФНИ МОН, договор No. ДН-02-2/13.12.2016

Апробация на резултатите

Резултатите, включени в дисертацията, са получени в съавторство с

- Илия Буюклиев [P2], [P4], [P8], [P9], [P7], [D2], [D4], [D5], [D6] ;
- Стефка Буюклиева [D1];
- Александра Стоянова [P3], [D3];
- Стефка Буюклиева и Александра Стоянова [P1];
- Илия Буюклиев и Александра Стоянова [P5];
- Стефка Буюклиева и Илия Буюклиев [P6], [D7].

Резултатите са публикувани или са приети за публикуване в международни научни списания:

- *Electronic Notes in Discrete Mathematics* [P6];

  в сборници от конференции:

- *Proceedings of the Forty Fourth Spring Conference of the Union of Bulgarian Mathematicians SOK*, Камчия, 2015 [P2];
- *Proceedings of XII International Conference ETAI*, Ohrid, Macedonia 2015, [P3];
- *Proceedings of 25 Years Faculty of Mathematics and Informatics*, Велико Търново, 2015 [P4];
- *Proceedings of 7th International Conference Information Technologies and Education Development*, Zrenjanin, Serbia, 2016 [P5];
- *Proceedings of PASCO 2017*, Kaiserslautern, Germany, 2017 [P7];

  в сборници:

- *UGD-Stip Yearbook of Faculty of Computer Science*, Stip, Macedonia, 2013 [P1];

  или се редактират в:

- *Scalable Computing: Practice and Experience* [P8];

Резултати от дисертацията са докладвани пред:

1. *Ежегоден национален семинар по теория на кодирането* (който в момента носи името "Стефан Додунеков"), Велико Търново, 2013 - 2015.
2. *XII International Conference ETAI*, Ohrid, Macedonia, 2015.
3. *Семинар по "Математически основи на информатиката"*, организиран съвместно от ФМИ-ВТУ и ИМИ-БАН, Велико Търново, 2015.
4. *25 YEARS FACULTY OF MATHEMATICS AND INFORMATICS*, Велико Търново, 2015.
5. *Fifteenth International Workshop on Algebraic and Combinatorial Coding Theory*, Албена, България, 2016;

Глава 1

Основни понятия за булеви и векторни булеви функции. Паралелно програмиране с CUDA.

В началото на тази глава са представени основни понятия, свързани с теорията на булевите и векторните булеви функции. След това са описани принципите на паралелното програмиране, паралелната изчислителна платформа и програмния модел CUDA, които използваме при разработването на паралелни програми.

1.1 Булеви функции

Булевите функции са едни от основните обекти в дискретната математика и свързаните с нея дисциплини.

Да разгледаме множество с два елемента, които условно означаваме с 0 и 1. За самото множество се използват различни означения, от които най-разпространени са $J_2 = \{0, 1\}$ в дискретната математика [2]. В алгебрата полето с най-малък брой елементи е $\mathbb{F}_2 = GF(2) = \{0, 1\}$, където 0 е нулев елемент, 1 е единичен елемент, и $1 + 1 = 0$ (т.е. събирането е по модул 2). Множеството от всички наредени n-орки J_2^n в дискретната математика е наречено *n-мерен булев куб* [2]. Можем да разглеждаме тези n-орки и като вектори в n-мерното линейно пространство $\mathbb{F}_2^n$. Тук ще използваме $\mathbb{F}_2$ и $\mathbb{F}_2^n$, където $n \geq 1$ е цяло число.

Съществува взаимно еднозначно съответствие (биекция) между $\mathbb{F}_2^n$ и мно-

жеството от цели числа $\{0, 1, 2, \ldots, 2^n - 1\}$, дефинирано по следния начин:

$$\overline{v} = (v_1, v_2, \ldots, v_n) \mapsto v_1 2^{n-1} + v_2 2^{n-2} + \cdots + 2v_{n-1} + v_n$$

и обратно, на числото $a \in \mathbb{Z}$, $0 \leq a \leq 2^n - 1$, съответства векторът $\overline{a} \in \mathbb{F}_2^n$, получен от двоичния запис на числото a с добавени нули отляво (ако $a < 2^{n-1}$).

Както вече споменахме, събирането в $\mathbb{F}_2$ е събиране по модул 2, което често се означава със знака $\oplus$. Затова и тук ще използваме това означение. Съответно за събирането на вектори имаме

$$v \oplus w = (v_1 \oplus w_1, v_2 \oplus w_2, \ldots, v_n \oplus w_n), \quad v, w \in \mathbb{F}_2^n.$$

Евклидовото скаларно произведение на вектори, се дефинира с

$$\langle v, w \rangle = v_1 w_1 \oplus v_2 w_2 \oplus \cdots \oplus v_n w_n \in \mathbb{F}_2, \quad v, w \in \mathbb{F}_2^n.$$

Дефиниция 1.1. *Булева функция* на n променливи f е изображение от $\mathbb{F}_2^n$ във $\mathbb{F}_2$, където $\mathbb{F}_2 = \{0, 1\}$ е полето с 2 елемента или всяка функция:

$$f : \mathbb{F}_2^n \to \mathbb{F}_2.$$

Една булева функция на n променливи може да бъде представена по различни начини: като вектор (таблица на истинност), като полином (алгебрична нормална форма), с "инфиксен" запис (виж [2]), при който се използват конюнкция, дизюнкция, отрицание и други логически операции, и т.н. Две от най-естествените представяния на булева функция на n променливи, които се използват за целите на криптографията са чрез таблица на истинност и алгебрична нормална форма.

Множеството на всички булеви функции ще означаваме с $\mathcal{F}$, а на булевите функции на n променливи с $\mathcal{F}_n$.

1.1.1 Таблица на истинност

Таблицата на истинност се дефинира като вектор, чиито координати са стойностите на f за всички n-мерни двоични вектори от $\mathbb{F}_2^n$.

Нека $v = (v_1, v_2, \ldots, v_n)$ и $w = (w_1, w_2, \ldots, w_n)$ са вектори от $\mathbb{F}_2^n$. Въвеждаме стандартната лексикографска наредба в $\mathbb{F}_2^n$, т.е. $v \prec w$, ако съществува $k \in \{1, 2, \ldots, n\}$, такова че $v_i = w_i$ за $i = 1, \ldots, k-1$, и $v_k < w_k$.

Пример 1.1. В $\mathbb{F}_2^3$ имаме

$$(000) \prec (001) \prec (010) \prec (011) \prec (100) \prec (101) \prec (110) \prec (111).$$

Лесно се вижда, че ако a и b са цели числа, $0 \leq a, b < 2^n$, то $\overline{a} \prec \overline{b} \iff a < b$. Това ни дава възможност на всяка булева функция да съпоставим еднозначно двоичен вектор $f \to v_f = (v_0, v_1, \ldots, v_{2^n-1}) \in \mathbb{F}^{2^n}$, където $v_a = f(\overline{a})$, $a = 0, 1, \ldots, 2^n - 1$.

Дефиниция 1.2. Двоичният вектор, който представя булевата функция f по описания по-горе начин, се нарича *таблица на истинност* за тази функция, или Truth Table (TT). Ще го бележим с v_f или с $TT(f)$, а с $[f]$ ще означаваме същия вектор, но разгледан като вектор-стълб (матрица с размер $2^n \times 1$).

Пример 1.2. Нека $f : \mathbb{F}_2^3 \to \mathbb{F}_2$ е зададена с $f(x_1, x_2, x_3) = x_1x_2x_3 \oplus x_2x_3 \oplus x_1 \oplus 1$. Тъй като

$$f(000) = 1, f(001) = 1, f(010) = 1, f(011) = f(100) = f(101) = f(110) = f(111) = 0,$$

то $TT(f) = (11100000)$.

Твърдение 1.1. [2] *Броят на булевите функции $f : \mathbb{F}_2^n \to \mathbb{F}_2$ е 2^{2^n}.*

Доказателство: Изображението $f \mapsto v_f$, което на всяка булева функция съпоставя нейната таблица на истинност TT, е биекция между множествата $\mathcal{F}_n$ и $\mathbb{F}_2^{2^n}$. Следователно $|\mathcal{F}_n| = |\mathbb{F}_2^{2^n}| = 2^{2^n}$. □

Когато броят n на входните променливи расте, броят на функциите драстично се увеличава.

Дефиниция 1.3. *Тегло по Хеминг* $\mathrm{wt}(f)$ или $w_H(f)$ е броят на наредените n-орки, за които стойността на булевата функция f е 1, следователно:

$$w_H(f) = \mathrm{wt}(f) = \sharp\{x \in \mathbb{F}_2^n \ : \ f(x) = 1\}.$$

Една булева функция на n променливи се нарича *балансирана*, ако теглото ѝ е 2^{n-1}.

Като имаме предвид, че теглото (по Хеминг) на двоичен вектор е броят на ненулевите му координати, имаме $\mathrm{wt}(f) = \mathrm{wt}(v_f)$. Под тегло $\mathrm{wt}(f)$ на функцията f ще разбираме теглото на вектора $TT(f)$.

Аналогично дефинираме:

Дефиниция 1.4. *Разстояние по Хеминг* $d_H(f, g)$ или $d(f, g)$ между булевите функции f и g се нарича:

$$d_H(f, g) = d(f, g) = \mathrm{wt}(f \oplus g) = \sharp\{x \in \mathbb{F}_2^n \ : \ f(x) \neq g(x)\}$$

Разстоянието (по Хеминг) между два вектора е броят на координатите, в които те се различават, следователно $d_H(f,g) = d_H(v_f, v_g)$. Под разстояние между две функции ще разбираме разстоянието между съответните таблици на истинност.

Дефиниция 1.5. *Носител (support)* на двоичния вектор $v = (v_1, \ldots, v_s)$ се нарича множеството $supp(v) = \{i : v_i \neq 0\}$. Аналогично, носител на булевата функция f е множеството от всички наредени n-орки, за които стойността на f е 1, следователно:

$$supp(f) = \{x \in \mathbb{F}_2^n \ : \ f(x) = 1\}.$$

Очевидно теглото на булева функция е равно на мощността на нейния носител, или $\mathrm{wt}(f) = \sharp supp(f)$.

С всяка булева функция f се асоциира функцията $(-1)^f = 1 - 2f$, чиито стойности са от множеството $\{-1, 1\}$. Множеството от стойности на функцията $(-1)^f$ се записват във вектор, който се нарича Polarity Truth Table (PTT) и се бележи с $PTT(f)$, а съответният вектор-стълб с $[(-1)^f]$. По-нататък използваме вектор-стълб $[(-1)^f]$ за да изчислим Walsh спектъра [11] на булева функция f.

1.1.2 Алгебрична нормална форма

Друго естествено представяне на булевата функция f е като двоичен полином на n променливи, в който всеки едночлен е произведение на променливи от нулева или първа степен. Това представяне е също еднозначно определено и е известно като алгебрична нормална форма (Algebraic Normal Form, ANF) [11].

Означаваме с x^u едночлените $x_1^{u_1} x_2^{u_2} \ldots x_n^{u_n}$, където $u \in \mathbb{Z}$, $0 \leq u \leq 2^n - 1, \bar{u} = (u_1, u_2, \ldots, u_n) \in \mathbb{F}_2^n$. Тогава алгебричната нормална форма на f е полинома

$$f(x) = f(x_1, x_2, \ldots, x_n) = \bigoplus_{u=0}^{2^n-1} a_n x^u. \tag{1.1}$$

Алгебричната нормална форма на булева функция f се нарича още полином на Жегалкин [67]. Степента на $ANF(f)$ наричаме алгебрична степен (algebraic degree) $deg(f)$ на булевата функция f,

$$deg(f) = max\{wt(\bar{u}) | a_u = 1\}, \text{ където } f(x) = \bigoplus_{u=0}^{2^n-1} a_n x^u.$$

Нека $\mathcal{FP}_n$ е множеството от всички двоични полиноми на n променливи, в които всеки едночлен е произведение на променливи от нулева и първа степен.

Теорема 1.2. *Всяка булева функция може да се представи еднозначно като двоичен полином от* $\mathcal{FP}_n$.

Доказателство: Можем да разглеждаме $\mathcal{FP}_n$ като линейно пространство с базис

$$1,\ x_1, \ldots, x_n,\ x_1x_2, \ldots, x_{n-1}x_n,\ \ldots,\ x_1x_2\cdots x_n.$$

Броят на тези едночлени е

$$1 + n + \binom{n}{2} + \binom{n}{3} + \cdots + 1 = 2^n.$$

Тъй като елементите на $\mathcal{FP}_n$ са всички линейни комбинации на базисните едночлени с коефициенти 0 и 1, то техният брой е 2^{2^n}, точно колкото е и броят на всички булеви функции от $\mathcal{F}_n$. Следователно $\mathcal{FP}_n \subseteq \mathcal{F}_n$, $|\mathcal{FP}_n| = |\mathcal{F}_n|$ и оттук $\mathcal{FP}_n = \mathcal{F}_n$. □

Очевидно е, че алгебрична нормална форма може да бъде свързана с двоичния вектор $(a_0, a_1, ..., a_{2^n-1}) \in \mathbb{F}_2^{2^n}$, чиито координати са коефициентите в представянето (1.1) при лексикографска наредба на n-орките $(u_1, u_2, \ldots, u_n)$. Ще означим този вектор-стълб с $[A_f]$.

В Глава 2 ще разгледаме връзката между TT и ANF и ще представим бърз алгоритъм за изчисление на таблицата на истинност $TT(f)$ ($[f]$) ако знаем $[A_f]$ и обратно.

1.1.3 Криптографски свойства на булевите функции

В тази подраздел дефинираме най-съществените криптографски свойства на булевите функции.

Булевите функции с алгебрична степен 1 играят специална роля в нашите изследвания затова представляваме следната дефиниция.

Дефиниция 1.6. Булева функция от вида $f(x) = a_0 \oplus a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n$ се нарича *афинна*. Ако $a_0 = 0$, т.е. $f(x) = a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n$, където $a = (a_1, a_2, \cdots, a_n)$ и $x = (x_1, x_2, \cdots, x_n)$, функцията е линейна.

Твърдение 1.3. *Булевата функция* f *е афинна тогава и само тогава, когато* $deg(f) \leq 1$.

Твърдение 1.4. *Броят на афинните булеви функции на* n *променливи е* 2^{n+1}, *а на линейните* 2^n.

От криптографска гледна точка важно свойство за една булева функция f е нейната нелинейност, която е свързана с разстоянието от f до всяка линейна функция и намирането на спектъра на *Walsh* [11].

Дефиниция 1.7. Нелинейност $nl(f)$ на булевата функция f е минималното Хемингово разстояние от f до афинните функции:

$$nl(f) = \min\{d_H(f, g) \mid g \ - \ \text{афинна функция}\}.$$

Твърдение 1.5. $nl(f) = 0 \iff f$ *е афинна функция.*

Теорема 1.6. [11] Границата на Парсевал (Parseval's relation)*:* $nl(f) \leq 2^{n-1} - 2^{n/2-1}$.

За използваните в криптографията булеви функции $nl(f)$ трябва да достига или да е близо до тази граница, за да е защитена системата от атаки чрез линейни апроксимации, корелационни атаки, и др.

Дефиниция 1.8. Булева функция, чиято нелинейност достига границата на Парсевал, се нарича бент-функция.

Необходимо условие за съществуване на бент-функции на n променливи е числото n да е четно, т.к. за нечетни n $2^{n-1} - 2^{n/2-1}$ въобще не е цяло число.

Дефиниция 1.9. Реалното число $\epsilon = \frac{1}{2^n} nl(f) - \frac{1}{2}$ се нарича отклонение от нелинейността (bias of nonlinearity) за булевата функция f.

За да дефинираме още параметри на булевите функции, които определят някои техни криптографски свойства, използваме трансфомация на Walsh.

Дефиниция 1.10. Трансформацията на Walsh (Hadamard, Walsh-Hadamard, Walsh-Fourier) f^W на булевата функция f, наричаме функцията $f^W : F_2^n \to \mathbb{Z}$, дефинирана чрез:

$$f^W(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus \langle a, x \rangle} = 2^n - 2d_H(f, f_a),$$

където $a = (a_1, \ldots, a_n) \in \mathbb{F}_2^n$, $f_a(x) = a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n$ и $f(x) \oplus \langle a, x \rangle = f(x_1, x_2, \ldots, x_n) \oplus a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n = f(x) \oplus f_a(x)$. Стойностите на f^W наричаме Walsh коефициенти на булева функция f.

В сила е следното равенство

$$f^W(a) = \sum_{x\in\mathbb{F}_2^n}(-1)^{f(x)\oplus\langle a,x\rangle} = \sum_{x\in\mathbb{F}_2^n, f(x)=f_a(x)} 1 + \sum_{x\in\mathbb{F}_2^n, f(x)\neq f_a(x)} (-1)$$

$$= |\{x\in\mathbb{F}_2^n, f(x)=f_a(x)\}| - |\{x\in\mathbb{F}_2^n, f(x)\neq f_a(x)\}| = 2^n - 2d_H(f,f_a).$$

За всяка булева функция f и всеки вектор $a \in \mathbb{F}_2^n$ е в сила $-2^n \leq f^W(a) \leq 2^n$. Граничните стойности се достигат от функциите $f_a(x) = \langle a,x\rangle$ и $\overline{f}_a(x) = \langle a,x\rangle \oplus 1$, като $f_a^W(a) = 2^n$ и $\overline{f}_a^W(a) = -2^n$.

Ако $\overline{f}(x) = f(x)\oplus 1$, то

$$\overline{f}^W(a) = \sum_{x\in\mathbb{F}_2^n}(-1)^{\overline{f}(x)\oplus\langle a,x\rangle} = \sum_{x\in\mathbb{F}_2^n}(-1)^{1\oplus f(x)\oplus\langle a,x\rangle} = -\sum_{x\in\mathbb{F}_2^n}(-1)^{f(x)\oplus\langle a,x\rangle} = -f^W(a).$$

При лексикографска наредба на векторите от $\mathbb{F}_2^n$, можем да наредим Walsh коефициентите $f^W(a)$ и да ги разглеждаме като координати на вектор. Този вектор наричаме Walsh спектър на булевата функция f и означаваме с $[W_f]$ (може да се разглежда и като стълб). Спектъра на Walsh $[W_f]$ на булева функция f на n променливи получаваме като произведение на $PTT(f)$ с Адамарова матрица H_n от ред n (от тип Sylvester)[40],

$$[W_f] = H_n[(-1)^f].$$

Тази матрица има не само рекурсивна структура, но тя е от специален вид, който позволява много ефективно (*бътерфлай*) умножение.

Множеството $\{W_f(i), -2^n \leq i \leq 2^n\}$, където $W_f(i)$ е броят на Walsh коефициентите със стойност i, се нарича Walsh разпределение (Walsh distribution) на функцията f.

Дефиниция 1.11. Булевата функция f се нарича корелационно имунна от ред r (correlation immune), ако $f^W(a) = 0$ за $1 \leq wt(a) \leq r$. Ако функцията е и балансирана, тогава тя притежава свойството устойчивост (resiliency) от ред t (т.е. $f^W(a) = 0$ за $0 \leq wt(a) \leq t$).

Дефиниция 1.12. Линейност Lin(f) на булевата функция f е максималната стойност на абсолютните стойности на коефициентите на Walsh: $f^W(a)$:

$$Lin(f) = max\{|f^W(a)|a \in \mathbb{F}_n^2\}.$$

Твърдение 1.7. *$nl(f) = 2^{n-1} - \frac{1}{2}Lin(f)$, а отклонението от нелинейността е $\epsilon = \frac{Lin(f)}{2^{n+1}}$*

Доказателство: Тъй като $f^W(a) = 2^n - 2d_H(f, f_a)$, то

$$\begin{aligned} Lin(f) &= \max\{|f^W(a)|a \in \mathbb{F}_2^n\} \\ &= \max\{|2^n - 2d_H(f, f_a)|a \in \mathbb{F}_2^n\} \\ &= \max\{\max\{2^n - 2d_H(f, f_a)|d_H(f, f_a) \leq 2^{n-1}\}\}, \\ &\quad \max\{2d_H(f, f_a) - 2^n|d_H(f, f_a) > 2^{n-1}\}. \end{aligned}$$

От $d_H(f, f_a) = 2^n - d_H(f, 1 \oplus f_a)$ получаваме $2d_H(f, f_a) - 2^n = 2^n - 2d_H(f, 1 \oplus f_a)$, откъдето

$$\begin{aligned} Lin(f) &= \max\{2^n - 2d_H(f, g)|g \ - \ \text{афинна функция}, d_H(f, g) \leq 2^{n-1}\} \\ &= 2^n - 2\min\{d_H(f, g)|g \ - \ \text{афинна функция}\} \\ &= 2^n - 2nl(f). \end{aligned}$$

Следователно $nl(f) = 2^{n-1} - \frac{1}{2}Lin(f)$. □

Следствие на равенството на Парсевал $\sum_{a \in \mathbb{F}_2^n}(f^W(a))^2 = 2^{2n}$, дава че $Lin(f) \geq 2^{n/2}$ [11]. Очевидно е, че минималната линейност е в пряка връзка с максимална нелинейност.

Дефиниция 1.13. Автокорелационна трансформация (Autocorrelation Transform - ACT) на булевата функция f, наричаме функцията $r_f : \mathbb{F}_2^n \to \mathbb{Z}$, дефинирана чрез:

$$r_f(w) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus f(x \oplus w)}$$

където $w \in \mathbb{F}_2^n$.

Автокорелационната трансформация ACT дава информация за диференциалните свойства на булевата функция f. За всяка булева функция f и всеки вектор $w \in \mathbb{F}_2^n$ е в сила $-2^n \leq r_f(w) \leq 2^n$, и $r_f(0) = 2^n$. Стойностите на $r_f(w)$ наричаме спектрални автокорелационни (spectral autocorrelation) коефициенти на булевата функция f. При лексикографска наредба на векторите от $\mathbb{F}_2^n$, можем да наредим автокорелационните коефициенти $r_f(w)$ и да ги разглеждаме като вектор. Този вектор наричаме Автокорелационнен спектър (Autocorrelation Spectrum) на булевата функция f, а с $[r_f]$ ще означаваме същия вектор, но разгледан като вектор стълб (матрица с размер $2^n \times 1$).

Автокорелация на булева функция f, означено с $AC(f)$, е максималната абсолютна стойност на автокорелационните коефициенти или: $AC(f) = \max\{|r_f(w)| \mid w \in \mathbb{F}_2^n\}$.

Според теоремата на Wiener-Khintchine [7]:

$$[r_f] = 2^{-n}[H_n]([W_f])^2, \tag{1.2}$$

където $([W_f])^2$ е вектор от квадратите на коефициентите на Walsh (квадратите на координатите на вектора $[W_f]$).

1.2 Векторни булеви функции

Тук ще обърнем внимание на основните понятия, свързани с векторните булеви функции.

Дефиниция 1.14. Векторна булева функция (също наричана (n, m) S-box, кратко S-box или субституциона кутия) може да се разглежда като функция S, която на редица от n входни бита съпоставя редица от m изходни бита (в много случаи се задава с таблица):

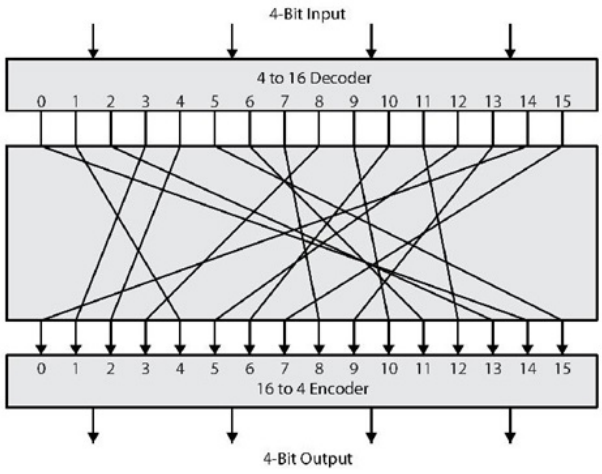
$$S : \mathbb{F}_2^n \to \mathbb{F}_2^m$$

Векторното представяне на векторните булеви функции е по следния начин $(f_1, f_2, \ldots, f_m)$, където f_i са булеви функции на n променливи, $i = 1, 2, \ldots, m$. Функциите f_i се наричат координатни функции на векторната булева функция. Те могат да се представят чрез техните таблици на истинност $TT(f_i)$ и по този начин векторната булева функция S може да се разглежда като матрица [12]:

$$G(S) = \begin{pmatrix} TT(f_1) \\ TT(f_2) \\ \cdots \\ TT(f_m) \end{pmatrix}.$$

Една векторна булева функция е обратима, ако $n = m$ и S е обратима функция. Тогава броят на променливите n наричаме размер на функцията.

Лема 1.8. *Една векторна булева функция е обратима (биективна) тогава и само тогава, когато $n = m$ и матрицата $G(S)$ поражда $[2^n, n]$ код, еквивалентен на разширения симплекс код $\overline{S}_n$ (разширен с нулева координата).*



Фигура 1.1: Обратима векторна булева функция която изобразява 4-битов вход в 4-битов изход

Както се вижда от лемата, важно значение в нашите конструкции на векторни булеви функции играе симплекс кода. Понеже използваме само двоични кодове, ще дефинираме само двоичен симплекс код. Двоичен линеен код с дължина n е всяко линейно подпространство на $\mathbb{F}_2^n$. Размерността на това подпространство наричаме размерност на кода. Матрица с k реда и n стълба, чиито редове образуват базис на подпространството, наричаме пораждаща матрица на линейния код. В някои случаи ще използваме и $s \times n$ матрици, $s > k$, които обаче имат ранг k, и чиито редове пораждат кода. Те не са пораждащи матрици в смисъла на дефиницията и затова за тях ще казваме, че пораждат кода (без да са пораждащи матрици). Минимално разстояние (минимално тегло) на линеен код е най-малкото измежду всички ненулеви тегла на кодови думи.

Нека $\overline{i}$ е двоичното представяне на цялото неотрицателно число i, разгледано като вектор от $\mathbb{F}_2^n$, $0 \leq i \leq 2^n - 1$. Линейният код с пораждаща матрица $G_n = (\overline{1}\ \overline{2}\ \cdots\ \overline{2^n - 1})$, където $\overline{i}$ е разгледан като вектор-стълб, се нарича n - *мерен симплекс код* и се бележи с S_n. Дължината на S_n е $2^n - 1$, а всичките му ненулеви кодови думи имат тегло 2^{n-1}. Да добавим нулев стълб в началото на матрицата G_n. Получената разширена матрица $\overline{G}_n$ поражда $[2^n, n, 2^{n-1}]$ линеен код, който ще наричаме разширен симплекс код и ще бележим с $\overline{S_n}$. Лесно се установява, че

$$\overline{G}_n = \begin{pmatrix} TT(x_1) \\ TT(x_2) \\ \cdots \\ TT(x_n) \end{pmatrix}.$$

За да получим код на Reed-Muller от първи ред, използваме матрицата

$$G(RM) = \begin{pmatrix} TT(1) \\ TT(x_1) \\ TT(x_2) \\ \cdots \\ TT(x_n) \end{pmatrix}.$$

Кодът с пораждаща матрица $G(RM)$ наричаме код на Reed-Muller от първи ред с параметри $[2^n, n+1, 2^{n-1}]$.

1.2.1 Криптографски свойства на векторните булеви функции

Векторните булеви функции представляват ключов елемент при дизайна на блоковите шифри. Те оформят нелинейния слой на блоковите шифри и поради това са много важни за сигурността на тези шифри. Съществуват добре изследвани критерии, който трябва да изпълнява шифърът, за да бъде устойчив на всички видове атаки.

Напомняме, че $\overline{S}_n = \langle TT(x_1), \ldots, TT(x_n) \rangle$ (Лема 1.8).

За изучаване на някои от криптографските свойства (линейност, нелинейност, алгебрична степен, автокорелация) на векторните булеви функции, трябва да вземем всички ненулеви линейни комбинации на координатните функции, означени с

$$S_b = b \cdot S = b_1 f_1 \oplus \cdots \oplus b_m f_m,$$

където $b = (b_1, ..., b_m) \in \mathbb{F}_2^m$. Тези линейни комбинации се наричат компонентни булеви функции.

Walsh спектър на векторните булеви функции се дефинира като множество от всички Walsh спектри на компонентните булеви функции. Всъщност Walsh спектъра на векторна булева функция, образува $(2^n \times 2^n)$ таблица на линейното приближение (Linear Approximation Table), която е свързана с линеен криптоанализ (Linear cryptanalysis) кратко означена с $LAT(S)$ [43, 27].

Линейност и нелинейност на векторна булева функция S се дефинира като:

$$Lin(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} Lin(b \cdot S), \qquad nl(S) = \min_{b \in \mathbb{F}_2^m \setminus \{0\}} nl(b \cdot S).$$

Автокорелационнен спектър на векторна булева функция е съвкупност от всички автокорелационни спектри на булевите компонентни функции и образува $(2^n \times 2^n)$ автокорелационна таблица AC(S). Автокорелация $AC(S)$ на

векторна булева функция дефинираме чрез:

$$AC(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} AC(b \cdot S).$$

Важно криптографско свойство на векторна булева функция представлява алгебричната степен. Някои автори дефинират $deg(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} deg(b \cdot S)$ [12], а други използват за $deg(S)$ минималната стойност измежду тези степени. По тази причина за алгебричната степен на векторна булева функция, дефинираме максимална и минимална алгебричен степен като:

$$deg(S) = \max_{b \in \mathbb{F}_2^m \setminus \{0\}} deg(b \cdot S), \qquad deg(S) = \min_{b \in \mathbb{F}_2^m \setminus \{0\}} deg(b \cdot S).$$

Алгебричната степен е максималната (минималната) стойност измежду компонентните булеви функции на векторната булева функция.

Difference Distribution Table (DDT), която е свързана с диференциалния криптоанализ (Differential cryptanalysis) [8, 27], се дефинира със следната формула:

$$DDT(S) = |\{x \in \mathbb{F}_2^n, \alpha \in \mathbb{F}_2^n \setminus \{0\}, \beta \in \mathbb{F}_2^m | S(x) \oplus S(x \oplus \alpha) = \beta\}|. \tag{1.3}$$

Диференциална еднаквост на $(n \times m)$ векторна булева функция при $n \geq m$, означен с δ, е друго важно криптографско свойство на векторните булеви функции, дефинирана чрез:

$$\delta = \max |\{DDT(S)\}|$$

където δ е максималната стойност на DDT(S) (като не се взема предвид стойностите на първата редица). Стойността на δ трябва да е колкото възможно по ниска. Добре е известно, че δ винаги получава четни стойности в интервала $[2^{n-m}, 2^n]$. Най-малката възможна стойност на δ за обратима векторна булева функция $(n = m)$ е 2 $(\delta \geq 2)$.

Връзката помежду δ и автокорелацията AC на векторните булеви функции е даден с израза $AC(S) = DDT(S) \times H_n$ [66], където H_n е $(2^n \times 2^n)$ Адамарова матрица от ред n [40], а $AC(S)$ е $(2^n \times 2^n)$ автокорелационата таблица на биективна векторна булева функция.

За представяне, определяне и изчисление на характеристиките на булевите функции са необходими ефективни алгоритми. Някои от тези алгоритми ще бъдат представени в Глава 2, а тяхната паралелна реализация в Глава 3 и 4.

1.3 Принципи на паралелното програмиране

В настоящето изследване се разглеждат булеви и векторни булеви функции или по-точно задачи, свързани с тези функции. Задачите свързани с представянето, дефинирането и изчисление на най-важните криптографски свойства на булевите функции и векторните булеви функции изискат ефективни алгоритми. За решаване на тези задачи при увеличаване на размера на входните данни е необходим по-голям изчислителен ресурс. За да намерим някои от криптографските свойства (линейност, автокорелация, алгебрична степен, диференциална еднаквост) е необходима реализация на ефективни алгоритми. От друга страна тези алгоритми са подходящи за паралелна реализация. Такава реализация днес е възможна чрез съвременните персонални компютри, тъй като те имат няколко процесора, процесори с повече ядра или имат графични платки, които включват процесор или процесори с повече ядра.

Паралелният компютър в общия случай представлява компютърна система с множество от изчислителни единици (процесори). Две основни категории на паралелни компютри са мултикомпютрите и централизирани мултипроцесори. Мултикомпютърът е паралелен компютър, изграден от множество компютри, свързани с комуникационна мрежа. Комуникацията между процесорите се осъществява чрез обмен на съобщения. Централизираният мултипроцесор (също наричан симетричен мултипроцесор) е високо интегрирана система, където всички процесори споделят достъпа до една глобална памет. Тази споделена памет поддържа комуникацията и синхронизацията между процесорите. Паралелното програмиране е програмиране на език, който позволява различни порции от изчисления да се изпълняват едновременно от различни процесори.

Компютърната система, която използваме за реализацията на алгоритмите има компютърна архитектура SIMD (Single Instruction Multiple Data), според класификацията на Flynn [21, 22, 23]. SIMD архитектурите се състоят от няколко независими процесора, които могат едновременно да изпълняват един поток от команди над различни потоци от данни. Приема се, че всички процесори изпълняват една и съща програма.

Наименованието SIMD не е изцяло изчерпателно, когато се описват паралелните архитектури. Често тези архитектури могат да бъдат класифицирани и според използвания паралелен модел за изпълнение (паралелизъм). Повече информация относно паралелните компютърни архитектури може да се намери в [1, 56].

За нашите изчисления ние използваме персонален компютър, който е с подходяща графична платка, която поддържа паралелно програмиране. За реализация на програмите използваме модела на хетерогенно паралелно прог-

рамиране. Обикновено хетерогенност в контекста на компютърните системи означава компютърна архитектура с различен набор инструкции, където главния процесор има една архитектура, а останалите друга архитектура. Тук ще се спрем на паралелна изчислителна платформа и програмен модел CUDA (Compute Unified Device Architecture), поради това, че програмите от настоящия труд са написани предимно с използване на този програмен модел.

1.3.1 Общи понятия за паралелните изчисления

Паралелно изчисление се използва от паралелните компютри, за да се намали необходимото време, за решаване на една изчислителна задача. Всъщност паралелното изчисление представлява модел на изчисление, при което тежка изчислителна задача се разделя на множество малки еднотипни задачи, които се решават едновременно (паралелно). Има различни начини, по които един проблем (задача) може да бъде разделен на по-малки. Тези начини са функционалното разделяне (functional decomposition) или разделянето по данни (data decomposition). Подходящото разделяне на задачата на подзадачи е от голямо значение за ефективно и оптимално използване достъпните компютърни ресурси. Това означава оптимизиране на изчислителния процес – максимална заетост на процесорите (load - balancing) при минимална комуникация (бърз достъп или обмен на данни).

Това не винаги е възможно и често се прави компромис между комуникация и изчисленията.

Следните термини са често използвани при описването и разработката на паралелните програми:

- **Скалируемост**, способността на една паралелна програма да се изпълнява по-бързо при увеличаване броя на процесорите за фиксиран размер на входните данни.

- **Разделяне/ Декомпозиция (Decomposition)**, начинът, по който задачата е разделена на по-малки части, като някои от тях или всички могат да се изпълняват паралелно (едновременно). Най-често това разделяне е по данни или по функционалност.

- **Задача**, проблемът, даден за решаване. В паралелния алгоритъм се прави декомпозиция на задачата.

- **Процеси (Processes)**, при последователното програмиране представляват програми в момента на изпълнение. При паралелните алгоритми (програми) имаме няколко последователни програми, които се изпълняват едновременно. Те са относително независими. Комуникират помежду си с цел решаването на обща задача. Тези програми ние ще наричаме нишки или процеси.

При CUDA главно се използва термина нишки.

- **Взаимодействие между процесите**, често при изпълнение на процесите се налага те да си предават информация.

- **Едновременност(Concurrency)**, отнася се до група процеси (нишки), които се изпълняват паралелно в даден момент. Една от основните цели при проектирането на паралелни програми е да се увеличи (доколкото е възможно) броя на паралелните процеси (нишки).

- **Паралелизъм (Parallelism)**, начинът или подходът за разпаралелване на програмата в зависимост от нейното разделяне. Например разделянето може да бъде по данни (Data parallelism) или функционално(function parallelism).

- **Гранулярност (Granularity)**, свързва се с размера на подзадачите, на които е разделен проблема. Ако разделянето е на много на брой малки подзадачи се нарича фина гранулация (fine-grained), ако е на малък брой големи подзадачи тогава е груба гранулация (coarse-grained).

- **Паралелна парадигма**, паралелния програмен модел. Най-общо този модел може да се отнася или до взаимодействие между процесите или до разделянето на проблема.

- **Латентност (Latency)**, отнася се до времето, необходимо на едно съобщение да достигне до местоназначението си при комуникация между процесите. Тя е пряко свързана със скоростта на изпълнение на паралелната програма. В латентността се включва и потвърждението, че съобщението е пристигнало.

- **Производителност (Performance)**, ефективността на една изчислителна машина и се характеризира с броя на операциите с плаваща запетая за секунда. Например 1 милиард извършени операции (най-често умножения с плаваща запетая) за една секунда са равни на 1 gigaflop (от англ. flop - floating-point operation).

1.3.2 GPU изчислителен модел

Използването на модерните графичнии процесори (Graphics Processing Unit - GPU) стана много популярно за изследователска работа. Това се дължи на тяхната способност за масивни паралелни изчисления. Съвременните графични процесори са много повече от ефективен и специализиран процесор за графични приложения. Тяхната силно паралелна структура ги прави по-ефективни от общо предназначените централни процесори за алгоритми, които обработват паралелно големи блокове от данни [33, 34]. В сравнение с централния процесор с повече ядра, новото поколение на графични процесори еволюира в изключително паралелен, многонишков (multithreaded), много-ядрен (manycore) процесор с огромна изчислителна мощност и много висока пропускателна способност

на паметта. Поради което те имат приложение в много области [34, 39, 53].

Графичните процесори са предназначени за ефективно изпълнение на хиляди нишки едновременно на колкото се може повече процесори за всеки един момент. Това едновременно стартиране на много нишки, позволява изпълнение на различни задачи от графичните процесори, докато данните се извличат или съхраняват в глобалната памет на графичната платка. Това също така осигурява скалируемост на GPU изчислителния модел [16].

Един от начините да разберем разликата между централния процесор и графичния процесор е да сравним как те обработват задачите. Централният процесор включва няколко ядра, оптимизирани за последователна обработка. Графичния процесор има масивна паралелна архитектура, която включва хиляди ефективни малки ядра, предназначени за обработка на множество задачи едновременно. Тези архитектура на графичните процесори, която поддържа хиляди нишки може значително да ускори изпълнението на определени програми в сравнение с CPU реализираните програми.

Традиционните CPU процесори имат латентно-ориентиран дизайн с мощен ALU, голяма кеш памет, сложен контрол и т.н. Тяхното предназначение е намаляване на времето за изпълнение на последователната програма с избягване на латентно-ориентирани задачи (когато е възможно). Техники като out-of-order изпълнение, speculative изпълнение и сложна кеш памет позволяват да се сведе до минимум латентността. От друга страна GPU процесорите имат агресивен throughput-ориентиран дизайн с малка кеш памет, опростен контрол, енергийно ефективни ALUs и изисква огромен брой на нишки, за да толерира латентността. Най-общо казано throughput-ориентираните процесори разчитат на три основни архитектурни характеристики: опростени ядра, масовата хардуерна многонишковост (multithreading) и SIMD изпълнение. За скриване на латентността при често движение на данните и постигане на пълна използваемост (utilization), GPU обикновено изисква хиляди нишки и големи масиви от данни.

1.4 CUDA платформа

През 2006 г. NVIDIA представи CUDA (Compute Unified Device Architecture) [17]. CUDA представлява паралелна изчислителна платформа и програмен модел с общо предназначение. Софтуерната среда на CUDA позволява на програмистите да пишат приложения на език от високо ниво. CUDA има различни програмни интерфейси, може да се използва в широк спектър от приложения и се подържа от определен брой архитектури на графични процесори [16, 17]

Тази платформа дава възможност за драстично увеличение на производителността на компютъра с използване на силата на графичния процесор (GPU).

В зависимост от архитектурата, CUDA ядрата са организирани в множество SM (Streaming Multiprocessor) с йерархия на паметта съдържа: набор от регистри, константи, texture кеш памет, споделена памет и глобална памет (global memory). Всеки SM сдържа няколко SPs (Streaming Processors), няколко SFU (Special Function Unit – използвани за специални функции като sin, cosin и т.н.). Общо прието наименование за SP е CUDA ядро. Всяко едно SP включва няколко ALUs и FPUs (Floating Point Unit). За всеки един цикъл, всяко ядро изпълнява една и съща инструкция над различни данни (SIMD), а комуникацията между SMs се осъществява чрез глобалната памет.

CUDA платформата е насочена към един клас на приложения, чиято контролна част се изпълнява на централния процесор. Той използва един или повече NVIDIA графични процесора. Полезността на този паралелен модел е, че подзадачите се изчисляват самостоятелно от група нишки на графичния процесор без намеса на централния процесор, като по този начин се получава оптимална полза от паралелния графичен хардуер.

Термините, които се използват са хост (host) за централния процесор, хост памет (оперативната памет), устройство (device) за графичния процесор, памет на устройството (глобалната памет на графичната платка) и паралелна функция (kernel).

Различните хардуерни архитектури на *GPU* са номерирани според изчислителните възможности (Compute Capability) [16].

1.4.1 CUDA програмен модел

Програмният модел CUDA ни осигурява възможност за програмиране на централния процесор и графичния процесор (разширение за хетерогенно програмиране). Тази програма, започва с изпълнение на хоста и има разширение, което позволява паралелно изпълнение на графичната платка. CUDA представлява програмен модел които комбинира добрите особености на традиционният (последователен) и паралелния програмен модел.

CUDA C [16] по същество е C/C ++ език с няколко разширения и runtime библиотека, което позволява изпълнение на паралелни функции на GPU. Като програмен интерфейс, CUDA C е програмен интерфейс близък по синтаксис на C, но концептуално и семантично е съвсем различен от C.

CUDA програмен код. Предназначение на NVCC.

Програмния код на CUDA приложенията сдържа съвкупност от конвен-

ционален C/C++ хост код и функции предназначени за устройството. CUDA C при компилиране се компилира, като *nvcc* разделя функциите за устройството от хост кода. След това се компилират функциите предназначени за устройството, а за хост кода се използва наличния C/C++ хост компилатор (gcc и g++ за Linux, clang и clang++ за Mac OS X и cl.exe за Windows). В етап на свързването се добавят специфични CUDA runtime библиотеки за поддръжка на конкретни GPU манипулации.

Компилатора nvcc приема набор от конвенционални опции, като дефиниране на макроси, включване на библиотеки и управляване на процеса на компилация. Стъпките, които не изпълняват от CUDA компилатора биват препращани към C++ хост компилатора (подържан от nvcc), така че nvcc превежда опциите във вариант разбираем за хост компилатора.

Обработката на потока на данните

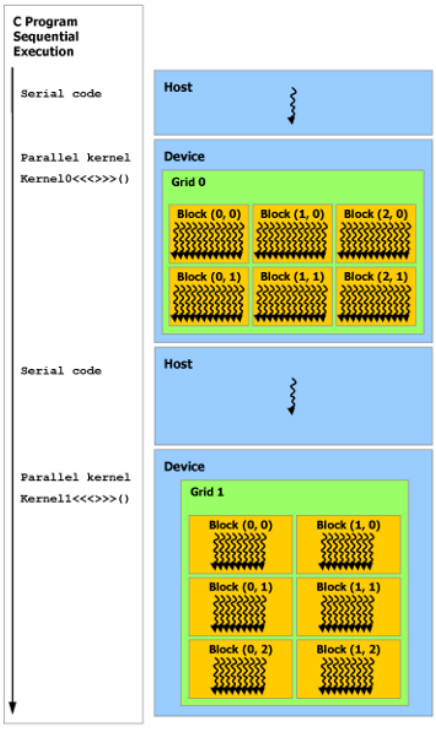
Обработката на потока на данните се осъществява на няколко стъпки. На най-високо ниво имаме главен процес, който се изпълнява на централния процесор и има следните действия: инициализира графичната карта; заделя оперативна памет на хоста (`malloc()`) и глобална памет на устройството (`cudaMalloc()`.); копира данните от оперативната памет на хоста в глобалната памет на устройството (`cudaMemcpy(, , ,cudaMemcpyHostToDevice)`); зарежда паралелния код; стартира определен брой инстанции за изпълнение на паралелни функции на устройството; копира данните от глобалната памет в оперативната памет (`cudaMemcpy(, , ,cudaMemcpyDeviceToHost)`); освобождава заетата памет (`cudaFree()`) и прекратява програмата.

Хетерогенно програмиране

Както е илюстрирано на (Фигура 1.2) при хетерогенно програмиране, програмния модел CUDA приема, че нишките се изпълняват на физически отделно устройство, което работи като копроцесор на хоста, който изпълнява C/C++ програма. Всъщност паралелните функции се изпълняват на графичния процесор, а остатъкът от C програмата на централния процесор.

Паралелни функции (Kernels)

Паралелните функции представляват програмни единици, наречени *kernels*. Множество нишки изпълняват паралелни функции, които извършват изчисления над поток от данни. Нишките представляват процес, който изпълнява поредица от независими програмни инструкции и е инстанция на паралелната функция. Създаването и унищожението на нишка почти не изисква ресурс (време за създаване и унищожаване). Нишките са организирани в бло-



Фигура 1.2: Илюстрация хетерогенно програмиране

кове (`blocks`). Блокът представлява набор от нишки, които могат да комуникират и да се синхронизират. Настоящите графични процесори поддържат до 1024 нишки (512 за по-старите) в блок. Всеки блок се изпълнява от един SM, а един SM може да изпълнява няколко блока едновременно, което зависи от специфичния GPU хардуер [16]. Преди изпълнение на паралелната функция е необходима конфигурация, която се задава с три ъглови скоби `<<< ... >>>`. В скобите се дефинира броя на блоковете и броя на нишките за блок. Броя на блоковете и нишките в блок определят модела за изчисление наречен още грид (`grid`). Всички нишки изпълняват един и същ код (SIMT - Single Instruction, Multiple Threads [51]). Всяка нишка има индекс (`tID`), който се използва за изчисляване на адреси от паметта и за вземане на решения за управление. Ако има M нишки за блок, индекса на текущата нишка може да бъде изчислен от индекса на текущия блок (`blockIdx.x`) и броя на нишки в този блок (`threadIdx.x`) чрез формулата:

$$tID = threadIdx.x + blockIdx.x * M.$$

Дефиницията на грид се прави по следния начин:

$$mykernel <<< \text{blocks per grid}, \text{threads per block} >>> (...);$$

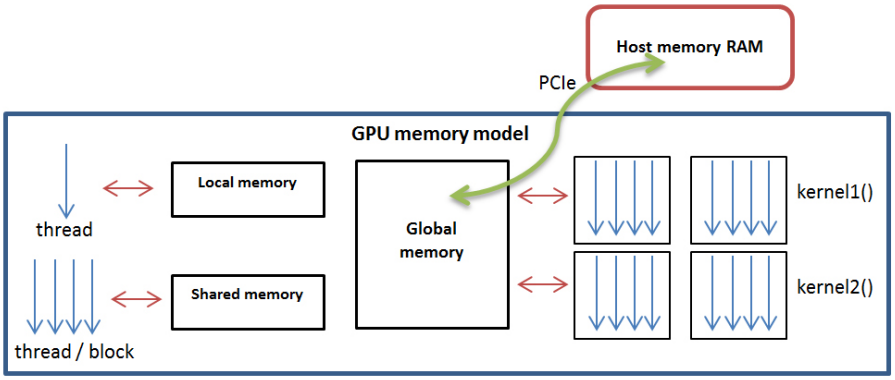
където `blocks per grid` преставлява броя на блокове, а `threads per block` броя на нишки за блок. Броя на нишките и на блоковете в грида определени с конфигурационния синтаксис `<<< ... >>>` може да бъдат от тип `int` или `dim3`. Може да дефинираме едномерен, двумерен или тримерен грид. Което може да направим по следния начин:

$$mykernel <<< dim3(bx, by, bz), dim3(tx, ty, tz) >>> (...);$$

където `dim3(bx, by, bz)` представлява грид от блокове, а броят на нишки за блок е дефиниран чрез `dim3(tx, ty, tz)`. Това позволява естествен начин за изчисления с вектори, матрици и т.н.

Модел на паметта

Скоростта при изпълнението зависи от правилното използване на паметта. Най-ниското ниво на паметта е най-бързо, но и най-скъпо и с ограничен размер. Регистрите са най-бързи, следвани от локалната памет, споделената памет и глобалната памет. Общ преглед на модела на паметта е представен на Фигура 1.3. Всяка нишка има достъп до локалната частна памет. Данните в споделената памет са достъпни до всички нишки от един и същ блок. Всички нишки от всички паралелни функции имат достъп до глобалната памет.



Фигура 1.3: Йерархията на паметта на графичната платка

Памет, която е само за четене и е достъпна за всички нишки през цялото време на изпълнение на програмата е паметта за константи и texture паметта.

Четенето на данни от паметта на константите е бързо почти колкото регистрите, а texture паметта е предназначена предимно за чисти графични приложения.

Синхронизация на нишките

За да имаме правилно изпълнение на паралелните функции, в някои случаи е необходима синхронизация на нишките във всеки блок. Инструкцията $\_\_syncthreads()$; създава "бариера"за синхронизация. Не е позволено нишките от един блок да преминават през тази точка (бариера), докато всички нишки не са я достигнали. Глобална синхронизация на всички нишките може се реализира с отделно изпълнение на паралелните функции или с Fast Barrier Synchronization [61].

1.4.2 CUDA архитектура

Архитектурата на NVIDIA графичните процесори е изградена от скалируем брой SM мултипроцесори.

Всеки SM организира и изпълнява групи от 32 нишки наречени контейнери (warps). Състоянието на контейнера може да бъде активно или временно неактивно, докато изчаква данни. Всеки контейнер се третира независимо. Обикновено всички нишки в контейнера изпълняват една и съща инструкция в даден момент.

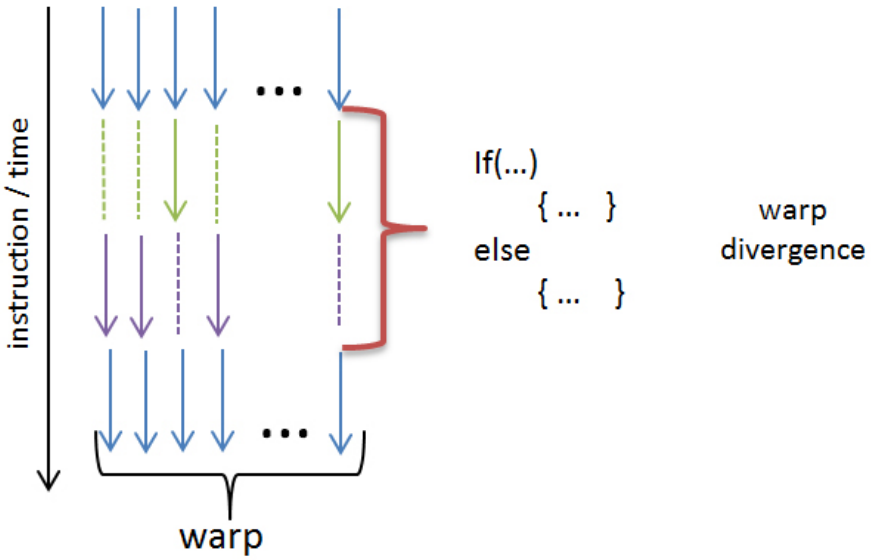
Какво се получава, ако различни нишки от един контейнер трябва да изпълняват различни инструкции? Това се нарича контейнерно отклонение (divergence). Подобна ситуация има при $if-then-else$ конструкцията (Фигура 1.4). При такава ситуация CUDA генерира коректен код, който се справя с това. При тези случаи има забавяне на изпълнението, тъй като всяка нишка изпълнява едно от двете разклонения и изчаква, докато другите нишки изпълняват другото разклонение. Общото време за изпълнение е сумата от времената на изпълнение на двете разклонения.

Броят на блоковете и контейнерите, които се изпълняват от един SM за дадена паралелна функция зависи от изискания ресурс (регистри, споделената памет) и наличния ресурс. Ако няма достатъчно наличен свободен ресурс, паралелната функция няма да започне изпълнение.

Общия брой контейнери в блок се изчислява по следния начин [17]:

$$ceil(\frac{T}{W_{size}}, 1)$$

където:



Фигура 1.4: Клон отклонението на контейнер

- T е броя на нишки за блок;
- W_{size} е размера на контейнера, който в нашия случай е равен на 32;
- $ceil(x, y)$ е равно на x закръглено до най-близкото кратно на y.

Общият брой на регистрите и общият размер на споделената памет, заделени за блок е документирано в CUDA Occupancy Calculator включен в CUDA платформата [16].

1.5 Паралелни алгоритми

В предните секции споменахме общите понятия на програмния модел CUDA. На практика паралелното изпълнение на програмата на CUDA започва, когато се извиква паралелна функция. При извикване на паралелна функция, както е описано по горе, всяка нишка прави собствено копие на самата функция и копие от всички променливи декларирани в нея. Единствената разлика между нишките е техния индекс, който се задава с променлива в копието на функцията. Изпълнението на всяка от нишките се определя от този индекс, така че той служи за управление на изпълнението на програмата, решение за управление, изчисляване на адрес на паметта, както и за паралелния модел на комуникация.

След като имаме обща представа за програмния модел и самото изпълнение на паралелните функции ще се насочим към по абстрактните понятия, относно стъпките и за самото проектиране. Паралелното програмиране може да се разгледа като процес на разделяне (decomposition) на проблема в добре дефинирани, координирани подзадачи, които могат да се решат с ефикасни математически изчислителни методи и добре познати алгоритми. Освен анализ на проблема, необходимо е да можем да го трансформираме в структурата, която ще има последователна част на изпълнение, част от която ще е възможно паралелно изпълнение и балансиране за получаване на по-добра ефикасност. Добрата декомпозиция на проблема осигурява избор и имплементиране на алгоритъм, който реализира подходящ компромис помежду паралелизма, изчислителната ефикасност и пропускателната способност на паметта.

Има три основни причини за използването на паралелни изчисления. Първата причина е решаване на даден проблем за по-малко време. Втората причина е използване на паралелно изчисление за решаване на големи задачи в рамките на определен период от време. Третата причина е получаване на по-добро (по-точно) решение на задача за определен период от време.

Някои алгоритми изискват по малко операции, но са по трудни за разпаралелване и обратно. В много от случаите се изиска повече памет за по-бързи изчисления. За избора на оптимална стратегия за изчисления обикновено се прави компромис между големината на използвания хардуерен ресурс и скоростта на изпълнение.

За да проектираме правилно паралелни алгоритми трябва да ги съобразим със:

- Компютърната архитектура: организация на паметта, пропускателната способност на паметта, добрите и лошите страни на компютърната архитектура SIMT, SPMD, SIMD и операциите с плаваща запетая срещу операциите на целочислените числа.

- Програмните модели и компилатори: паралелни модели на изпълнение, типове на достъпната памет, масиви, подробности за нишките.

- Алгоритмични техники: tiling, cutoff, scatter - gather, binning и други. Те имат голямо влияние върху скалируемост, производителността и пропускателната способност на паметта.

- Познание на задачата: математически методи и модели, детайли, област на прилагане и т.н. Самото познание на задачата осигурява прилагане на по-креативни алгоритмични техники.

Процеса на паралелно програмиране може да се раздели на четири стъпки: разделяне на проблема, избор на подходящ алгоритъм, имплементиране на програмен език и изчислителни настройки.

1.5.1 Разделяне на задача

Намирането на паралелизъм в големи изчислителни задачи често е концептуално ясен, но представлява предизвикателство на практика. Ключов момент е да се определи работата, която ще се реализира от всяка инстанция (CUDA нишка) при паралелното изчисление. При самото разделяне на проблема на подзадачи, които ще се изчисляват от нишките трябва да се обърне особено внимание на самото подреждане на нишките (threading arrangement). Има няколко модела на подреждане на нишките. Някои модели може да доведат до сходни нива на паралелно изпълнение, но може да имат и различна производителност за дадена хардуерна архитектура.

Реалните приложения често се състоят от множество модули, които работят заедно, което означава че трябва да се реши как да се организира работата. Освен това обемът на работата, може да варира драстично между тези модули. Тези модули могат да се реализират като отделни структурирани потоци от данни. Трябва да се реши кой поток си заслужава паралелно изпълнение, а за кой е по-добре да се изпълни от централен процесор.

Когато се върши разделяне на подзадачи в зависимост от тяхната големина имаме фина или груба гранулация и освен това подзадачите трябва да бъдат по възможност непресичащи. При разглеждания (CUDA) модел се изисква да имаме фина гранулация поради това, че имаме възможност за пускане на хиляди нишки.

Затова се извършава разделяне на данните (domain/data decomposition). Декомпозицията трябва да е съобразена с следното:

- Трябва планираният ресурс да е по-малък от наличния.

- Подзадачите требва да се еднакви или относително еднакви, за да има балансиран товар между нишките.

- Размерът на задачата да не указва влияние на размера на подзадачата. Увеличението на задачата да води само до увеличение на броя на подзадачите.

1.5.2 Оценка на производителността на паралелни алгоритми

При проектиране на паралелни алгоритми, от особено значение е възможността да се предвиди с приблизителна точност производителността им. Може да

се анализира времето на изпълнение на паралелния алгоритъм и също да се предвиди какво подобряване е възможно да се реализира с увеличаване на изчислителния паралелен ресурс. За подобен анализ ние използваме таблично представяне на времето на изпълнение на алгоритъма при различен брой процеси/нишки в зависимост от размера на задачата.

Поведението на алгоритъма може да се оцени и теоретично, чрез пресмятане на ускорението, капацитета на паметта и др.

Ускорение

Ускорение се дефинира със следната формула:

$$S_p = \frac{T_{(1,n)}}{T_{p(n)}}, \quad \text{където} \quad 1 \leq S_p \leq p \tag{1.4}$$

където имаме отношение помежду времето на изпълнение на последователен алгоритъм $T_{(1,n)}$ и времето за изпълнение на паралелен алгоритъм $T_{p(n)}$, постигнато при решаване на задача с размер n чрез използването на p процеса.

Операциите, които се изпълняват при паралелните алгоритми, могат да се разделят в три категории:

– Изчисления, които трябва да се изпълнят последователно;

– Изчисления, които могат да се изпълнят паралелно;

– Допълнителни разходи (parallel overhead), за комуникация и за излишни изчисления.

Закон на Amdahl [4]

Частта от работата, която се изпълнява паралелно ще определи нивото на ускорение на приложението, което се получава при паралелизацията. Закона на Amdahl е един от основните закони, които определят максималното постигнато ускорение. Той се базира на предположение, че се опитваме да решим проблем с фиксирана големина възможно най-бързо. Чрез него се дава горната граница от ускорението и може да се види ограничението му от последователната част на изпълнението му. Формулата за изчисление на максималното постигнато ускорение ψ на паралелен компютър, изпълняващ дадения алгоритъм е:

$$\psi \leq \frac{1}{f + (1-f)/p}$$

където с f е означена частта от алгоритъма която трябва да се изпълни последователно, а съответно останалата част $1-f$ може да бъде изпълнена паралелно, p е броя на процесорите, които изпълняват паралелния код. Поради това, че тук имаме хетерогенно програмиране и процесора на графичната платка няма

съща производителност като централния процесор, за p се взима ускорението, което се получава за частта от кода, която се изпълнява паралелно.

Този формула показва, че е необходимо последователната част да се направи колко се може по-малка, за да се постигне по-добро ускорение. Освен това тук не е взето предвид обработката на потока на данните (допълнителни разходи). Така на пример ако програмата губи 50% за обработката на потока данни (четене, записване) можем да получим до $2x$ пъти ускорение.

Честотната лента на паметта (Memory Bandwidth)

Честотната лента (bandwidth), показва скоростта на трансфера на данните и е един от най-важните фактори, влияещи на производителността. Изборът на паметта, в която се съхраняват данните и шаблоните на комуникация силно влияят на честотната лента.

За правилно измерване на производителността е полезно изчислението на теоретичната и действителна ефективност на честотната лента. Когато ефективната е много по-малка от теоретичната, алгоритъмът може да се оптимизира.

Теоретичната скорост на честотната лента изчисляваме чрез спецификацията на хардуера, като използваме следното равенство:

$$theoretical\ bandwidth = (MCR(Hz) \times MIW(bytes) \times X)/CR$$

където MCR(memory clock rate) честота на паметта, MIW (Memory Interface Width) ширина на интерфейса, $X = 2$ ако GDDR5 (double data rate) RAM, $CR = 10^9$ (convert result GB/s) за конвертиране резултата в GB/s.

Ефективното изчисление на честотната лента е когато стартираме алгоритъма върху определен поток от данни. За да я изчислим използваме следното равенство:

$$effective\ bandwidth = ((B_r + B_w)/10^9)/time$$

Тук ефективната честотна лента изчисляваме в GB/s. B_r е броя на байтове, които паралелната функция чете, B_w броя байтове (тип променливи) записани от паралелната функция и времето дадено в секунди.

Оценка на изчислителната пропускателна способност (Computational Throughput)

Друга важна метрика за оценка на производителността е изчислителната пропускателна способност (Computational Throughput). Обща мярка за изчислителната пропускателна способност е $FLOP/s$ (FLoating-point OPerations per

second) или $GFLOP/s$ (G-Giga префикс за 10^9). Събиране, изваждане, умножение или деление е един $flop$. Тази метрика показва брой на flops, поддържани от приложението.

Както теоретичната скорост на честотната лента, така и теоретичната изчислителната пропускателна способност зависи от спецификацията на хардуера.

За измерване на ефективната изчислителната пропускателна способност използваме следното равенство:

$$GFLOP/s \ _{Effective} = Total \ flops/(time \times 10^9) Gflops/sec$$

където ($Total \ flops$) е броя на $flops$, $time$ е време на работа на паралелната функция. Изчисляване на броя на $flops$ на паралелната функция е трудно. Поради това е прието използването на софтуерни инструменти (profiling tools) [55].

1.5.3 Ализиране на ефективни програми на CUDA

Писането на ефективни програми на CUDA се базира на три основни стратегии:

- Оптимизиране на паралелното изпълнение, за да се постигне максимална производителност.

- Оптимизиране (минимизиране) на времето, което всяка нишка губи за операции за достъп до данните като четене и записване. Това означава минимизиране на бавния трансфер на данни, или минимизиране на трансфера на данните помежду хоста и устройството. Добре е често използваните данни да се преместват в по-бърза памет като регистрите, локалната или споделената памет когато се използват за изчисления.

- Оптимизиране на използваемостта на инструкциите, за да се постигне максимална производителност. Различните математически инструкции заемат различно време. Трябва да се използват вградените инструкции, както и изчисляване с двойна точност.

1.6 Общи тенденции за бъдещо развитие на графичните платки като изчислителен ресурс

В тази част на кратко ще дадем обобщение на общите тенденции и развитието на Nvidia графичните платки. От самото представяне на CUDA и до днес тя има непрекъснато развитие. Това включва увеличение на изчислителния

ресурс и подобряване на програмния инструментариум с добавяне на нови и допълнителни функционалности.

Ако направим сравнение на различните поколения Nvidia графични платки Tesla K40, Tesla M40, Tesla P40 [26], които са от един и същ клас и са предназначени за високопроизводителни изчисления (High Performance Computing - HPC), можем да констатираме, че от поколение на поколение имаме значително увеличаване на изчислителния ресурс, размера и пропускателната способност на паметта, а с това и производителността. Също така, ако се разгледат предимствата на различните архитектури, според поколението, колкото то е по-ново имаме повече възможности и функционалности. Освен това, при последната архитектура Pascal (Tesla P40) [26] се използват нови технологии, като FinFET 16nm, която увеличава плътността (броя на транзистори), NVLink технология за високоскоростна връзка между графичните платки [65], HBM2 stacks (High Bandwidth Memory 2). Въвеждането на обединена памет (Unified Memory) предоставя значително подобряване на програмния модел, което означава, че имаме единствена памет за хоста и за устройството (това се поддържа от новите архитектури).

Очаква се за в бъдеще графичните платки, да имат все по-голяма и по-голяма изчислителната мощност. Това ще ни осигури ускоряване на паралелните изчисления. Само можем да предполагаме какво ще ни предостави следващото поколение (Volta).

1.7 Коментари

Начални изследвания, свързани с криптографските свойства, характеристики и класификация на векторни булеви функции (S-boxes), са представени в [P1].

Относно паралелното програмиране първоначално се спряхме на умножението на вектор по матрица, чиято паралелна реализация е тривиална. В публикацията [P3], която е докладвана на конференция [D3], е представена паралелна реализация на умножение на вектор по матрица, свързана със спектъра на Walsh, на което по-подробно ще се спрем в следващите глави.

Глава 2

Алгоритми за трансформации на булеви функции, чрез двоично представяне на целите неотрицателни числа

В тази глава се обсъждат някои трансформации на булеви функции, чието описание се прави много естествено чрез двоичното представяне на целите неотрицателни числа. Основните точки на тази глава са представяне на някои алгоритми с приложение в криптографията. На практика разглежданите алгоритми се реализират чрез умножение на вектор с матрица.

2.1 Въведение

Както споменахме в първа глава две от най-естествените представяния на булевите функции на n променливи са чрез таблица на истинност и алгебрична нормална форма. Това представяне се оказва много удобно за описание на някои трансформации и представяния на булеви функции и на алгоритми, свързани с тях. Тема на тази глава е един подход за приложение на двоичното представяне на целите неотрицателни числа в пресмятания с булеви функции.

Нека $S = \{0, 1, 2, \ldots, 2^n - 1\}$. На всяко число $u \in S$, $u = u_1 2^{n-1} + u_2 2^{n-2} + \cdots + u_{n-1} 2^1 + u_n$ можем да съпоставим еднозначно вектора $\overline{u} = (u_1, u_2, \ldots, u_{n-1}, u_n)$. С S_{set} бележим множеството $S_{set} = \{\overline{0}, \overline{1}, \overline{2}, \ldots, \overline{2^n - 1}\}$. Да дефинираме матрицата $S^{(n)}_{mat}$ по следния начин: $S^{(n)}_{mat} = \left(\overline{0}\ \overline{1}\ \cdots\ \overline{2^{n-1}}\right)^t$. Матрицата $S^{(n)}_{mat}$ може

да се зададе рекурсивно, т.к. е в сила следното представяне $S_{mat}^{(n+1)} = \begin{pmatrix} 0 & S_{mat}^{(n)} \\ 1 & S_{mat}^{(n)} \end{pmatrix}$.

Всяка булева функция f на n променливи еднозначно се дефинира с таблицата си на истинност $TT(f)$, която представлява вектор с координати стойността на f за стойности на променливите $(\overline{0}, \overline{1}, \ldots, \overline{2^{n-1}})$.

Друго естествено представяне на булевата функция f е като двоичен полином на n променливи, в който всеки едночлен е произведение на променливи от нулева или първа степен. Това представяне е също еднозначно определено и е известно като алгебрична нормална форма (ANF) [11].

Едната от целите на тази глава е да се опише и мотивира бърз алгоритъм за преобразуването на булева фунция от едното представяне в другото и обратно. От криптографска гледна точка важно свойство за една булева функция f е нейната нелинейност, която е свързана с разстоянието на f до афинните функции и намирането на спектъра ѝ на *Walsh*(виж [11]). Другата ни цел е представяне на ефективен алгоритъм в тази посока. На практика разглежданите алгоритми се реализират чрез умножение на вектор с матрица. Особеното е, че матриците имат не само рекурсивна структура (свързана с рекурсивната структура на матрицата $S_{mat}^{(n)}$), но тя е от специален вид, който позволява много ефективно (*бътерфлай*) умножение. Въпреки, че предложените алгоритми не са по-добри в порядък от досега известните, те са много по-компактни и много по-нагледни.

2.2 Алгоритъм за преобразуване на булеви функции

Нека булевата функция f е представена в ANF. Тогава тя се записва като сума на едночлени от вида $x_{i_1}x_{i_2}\cdots x_{i_k}$, $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, $1 \leq k \leq n$.

Всечки едночлен може да се дефинира по естествен начин чрез елементите на S_{set}. Да означим с $x^{(*u)}$ едночлена $M^u(x) = x_1^{u_1}x_2^{u_2}\ldots x_n^{u_n}$ където $u \in S_{set}$. Стойноста на $M^u(x)$ за $x = v$, $v \in S_{set}$, получаваме след като пресметнем $M^u(v) = v_1^{u_1}v_2^{u_2}\ldots v_j^{u_j}\ldots v_n^{u_n}$. Ако $u_j = 0$ или $u_j = v_j = 1$, j-тият множител не влияе на стойността на $M^u(v)$. Ако $u_j = 1, v_j = 0$, то $M^u(v) = 0$. Таблицата на истинност на $M^u(x)$ е равна на

$$\begin{pmatrix} M^u(\overline{0}) \\ M^u(\overline{1}) \\ \vdots \\ M^u(\overline{2^n-1}) \end{pmatrix} = \begin{pmatrix} \overline{0}^{*u} \\ \overline{1}^{*u} \\ \vdots \\ \overline{2^n-1}^{*u} \end{pmatrix} = \left(S_{mat}^{(n)}\right)^{*u}.$$

Нека булевата фнкция f е зададена с алгебричната си нормална форма $f(x) = f(x_1, x_2, \ldots, x_n) = f_0 x^{*\overline{0}} \oplus f_1 x^{*\overline{1}} \oplus \cdots \oplus f_{2^n-1} x^{*\overline{2^n-1}}$. ANF на f се задава еднозначно с вектора $\overline{f} = (f_0, f_1, \ldots, f_{2^n-1})$. Тогава за таблицата на истинност на f имаме:

$$\begin{pmatrix} f(\overline{0}) \\ f(\overline{1}) \\ \vdots \\ f(\overline{2^n-1}) \end{pmatrix} = \begin{pmatrix} \overline{0}^{*\overline{0}} & \overline{0}^{*\overline{1}} & \ldots & \overline{0}^{*\overline{2^n-1}} \\ \overline{1}^{*\overline{0}} & \overline{1}^{*\overline{1}} & \ldots & \overline{1}^{*\overline{2^n-1}} \\ & & \cdots & \\ \overline{2^n-1}^{*\overline{0}} & \overline{2^n-1}^{*\overline{1}} & \ldots & \overline{2^n-1}^{*\overline{2^n-1}} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \cdots \\ f_{2^n-1} \end{pmatrix} = A_n \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{2^n-1} \end{pmatrix},$$

$$A_n = \left((S_{mat}^{(n)})^{*\overline{0}} \ (S_{mat}^{(n)})^{*\overline{1}} \ \cdots \ (S_{mat}^{(n)})^{*\overline{2^n-1}} \right) = \left(\begin{pmatrix} 0 & S_{mat}^{(n-1)} \\ 1 & S_{mat}^{(n-1)} \end{pmatrix}^{*\overline{0}} \cdots \begin{pmatrix} 0 & S_{mat}^{(n-1)} \\ 1 & S_{mat}^{(n-1)} \end{pmatrix}^{*\overline{2^n-1}} \right).$$

Първата координата на векторното представяне на числата до $2^{n-1} - 1$ е 0, затова

$$((0 \ S_{mat}^{(n-1)})^{*\overline{0}} \ldots (0 \ S_{mat}^{(n-1)})^{*\overline{2^{n-1}-1}}) = ((1 \ S_{mat}^{(n-1)})^{*\overline{0}} \ldots (1 \ S_{mat}^{(n-1)})^{*\overline{2^{n-1}-1}}) = A_{n-1}.$$

Първата координата на векторите $\overline{i}$ за $i \geq 2^{n-1}$ е 1. Ако стойноста на x_1 е единица, тя не повлиява на стойността на едночлените, а ако стойноста на x_1 е нула стойността на едночлена е нула. Следователно

$$((1 \ S_{mat}^{(n-1)})^{*\overline{2^{n-1}}} \cdots (1 \ S_{mat}^{(n-1)})^{*\overline{2^n-1}}) = A_{n-1}, \ \ ((0 \ S_{mat}^{(n-1)})^{*\overline{2^{n-1}}} \cdots (0 \ S_{mat}^{(n-1)})^{*\overline{2^n-1}}) = 0.$$

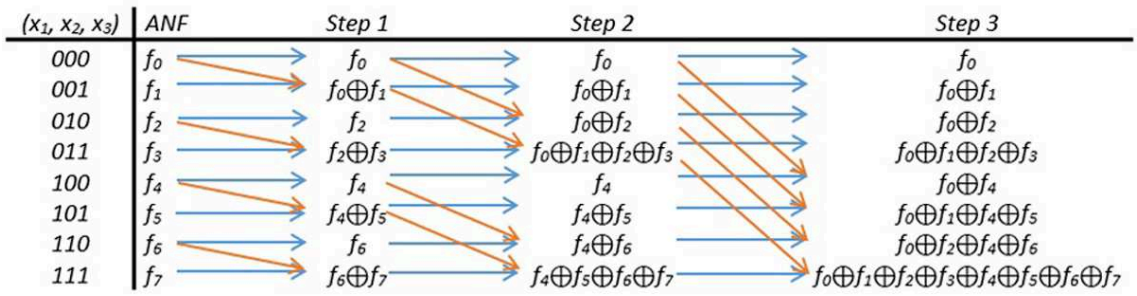
Тези равенства показват, че $A_n = \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix}$ и $A_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$,

$$A_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \quad A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix},$$

$$A_3 \cdot \overline{f}^t = \begin{pmatrix} f_1 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \\ f_0 \oplus f_1 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \\ f_0 \oplus\ 0 \oplus\ f_2 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \\ f_0 \oplus f_1 \oplus f_2 \oplus f_3 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \\ f_0 \oplus\ 0 \oplus\ 0 \oplus\ 0 \oplus f_4 \oplus\ 0 \oplus\ 0 \oplus\ 0 \\ f_0 \oplus f_1 \oplus\ 0 \oplus\ 0 \oplus f_4 \oplus f_5 \oplus\ 0 \oplus\ 0 \\ f_0 \oplus 0 \oplus\ f_2 \oplus\ 0 \oplus\ f_4 \oplus\ 0 \oplus f_6 \oplus 0 \\ f_0 \oplus f_1 \oplus f_2 \oplus f_3 \oplus f_4 \oplus f_5 \oplus f_6 \oplus f_7 \end{pmatrix}.$$

Матрицата A_n е двочна матрица с размери $2^n \times 2^n$ и детерминанта 1, така че $A_n \in \mathrm{SL}(2^n, \mathbb{F}_2)$. Лесно се пресмята, че $A_n^2 = I_{2^n}$ и $A_n^{-1} = A_n$. Затова можем да направим следното заключение: от ANF $\overline{f}$ на булева функция можем да намерим $TT(f)$ с умножение по A_n и обратно. Или $TT(f) = A_n\overline{f}$ и $\overline{f} = A_n \cdot TT(f)^t$. Трансформацията μ, която изобразява ANF на булева функция в таблица на истинност и обратно се нарича трансформация на Möbius. В литературата често срещано наименование на този алгоритъм е трансформация на Reed–Muller [31].

Нека разгледаме внимателно сумите в $A_3 \cdot \overline{f}^t$ от примера по-горе. Вижда се, че сумата от събираеми 1 и 2 на ред 2 се повтаря във всеки четен ред, а сумата от събираеми 1 и 2 на ред 1 се повтаря във всеки нечетен ред. Подобно повторение се забелязва и за ненулевите събираеми 3 и 4 и т.н. Това прави възможно умножението на матрица по вектор да се изпълни за n стъпки вместо 2^n чрез следната бътерфлай *Диаграма 1* (Фигура 2.1), като се избягват вече направените събирания.



Фигура 2.1: Диаграма 1: Бърза трансформация на Мьобиус

Верността на тези разсъждения в общия случай може да се види в [11]. Тази диаграма (Фигура 2.1) се реализира със следния Алгоритъм 2.1.

Algorithm 2.1 Fast Möbius Transform

Input: The Truth Table TT of the Boolean function f, with 2^n entries
Output: The Algebraic Normal Form ANF of the Boolean function f, with 2^n entries

```
j ← 1; ANF ← TT;
while (j < 2^n) do
    for i = 0 to 2^n − 1 do
        if ((i&j) == j) then
                    ANF[i] ← (ANF[i] ⊕ ANF[i − j])
     end for
    j ← 2 ∗ j;
end while.
```

2.3 Алгоритъм за пресмятане на Walsh спектър

Нека $f(x) = u_1x_1 \oplus u_2x_2 \oplus \cdots \oplus u_nx_n$ е линейна булева функция на n променливи. Ще използваме означението $u_1x_1 \oplus u_2x_2 \oplus \cdots \oplus u_nx_n = x^{(\oplus u)}$. Двоичният n-мерен вектор u еднозначно определя функцията $f(x)$ и затова ще я бележим и с $f^{(\oplus u)}(x)$. Стойността на $f^{(\oplus u)}(x)$ за вектора v се пресмята като $f^{(\oplus u)}(v) = u_1v_1 \oplus u_2v_2 \oplus \cdots \oplus u_nv_n$. Таблицата на истинност на $f^{(\oplus u)}(x)$ има вида

$$\begin{pmatrix} f^{(\oplus u)}(\overline{0}) \\ f^{(\oplus u)}(\overline{1}) \\ \vdots \\ f^{(\oplus u)}(\overline{2^n-1}) \end{pmatrix} = \begin{pmatrix} \overline{0}^{(\oplus u)} \\ \overline{1}^{(\oplus u)} \\ \vdots \\ \overline{2^n-1}^{(\oplus u)} \end{pmatrix} = \left(S_{mat}^{(n)}\right)^{(\oplus u)}.$$

Стойностите на функциите за $\overline{0}, \overline{1}, \ldots, \overline{2^n-1}$ образуват следната матрица:

$$H_n^+ = \begin{pmatrix} \overline{0}^{\oplus\overline{0}} & \overline{0}^{\oplus\overline{1}} & \ldots & \overline{0}^{\oplus\overline{2^n-1}} \\ \overline{1}^{\oplus\overline{0}} & \overline{1}^{\oplus\overline{1}} & \ldots & \overline{1}^{\oplus\overline{2^n-1}} \\ & & \cdots & \\ \overline{2^n-1}^{\oplus\overline{0}} & \overline{2^n-1}^{\oplus\overline{1}} & \ldots & \overline{2^n-1}^{\oplus\overline{2^n-1}} \end{pmatrix}$$

Следва, че

$$H_n^+ = \left(\left(S_{mat}^{(n)}\right)^{\oplus\overline{0}}, \left(S_{mat}^{(n)}\right)^{\oplus\overline{1}}, \ldots, \left(S_{mat}^{(n)}\right)^{\oplus\overline{2^n-1}}\right)$$

$$= \left(\begin{pmatrix} 0S_{mat}^{(n-1)} \\ 1S_{mat}^{(n-1)} \end{pmatrix}^{\oplus\overline{0}} \cdots \begin{pmatrix} 0 & S_{mat}^{(n-1)} \\ 1 & S_{mat}^{(n-1)} \end{pmatrix}^{\oplus\overline{2^{n-1}-1}} \begin{pmatrix} 0 & S_{mat}^{(n-1)} \\ 1 & S_{mat}^{(n-1)} \end{pmatrix}^{\oplus\overline{2^{n-1}}} \cdots \begin{pmatrix} 0 & S_{mat}^{(n-1)} \\ 1 & S_{mat}^{(n-1)} \end{pmatrix}^{\oplus\overline{2^n-1}} \right)$$

За матриците H_n^+ имаме:

$$((0\ S_{mat}^{(n-1)})^{\oplus\overline{0}} \ldots (0\ S_{mat}^{(n-1)})^{\oplus\overline{2^{n-1}-1}}) = ((1\ S_{mat}^{(n-1)})^{\oplus\overline{0}} \ldots (1\ S_{mat}^{(n-1)})^{\oplus\overline{2^{n-1}-1}}) = H_{n-1}^+,$$

$$((1\ S_{mat}^{(n-1)})^{\oplus\overline{2^{n-1}}} \ldots (1\ S_{mat}^{(n-1)})^{\oplus\overline{2^n-1}}) = \overline{H_{n-1}^+},$$

където матрицата $\overline{H_{n-1}^+}$, се получава от H_{n-1}^+, като единиците се сменят с нули а нулите с единици. От това следва, че:

$$H_n^+ = \begin{pmatrix} H_{(n-1)}^+ & H_{(n-1)}^+ \\ H_{(n-1)}^+ & \overline{H_{(n-1)}^+} \end{pmatrix}, H_1^+ = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, H_2^+ = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

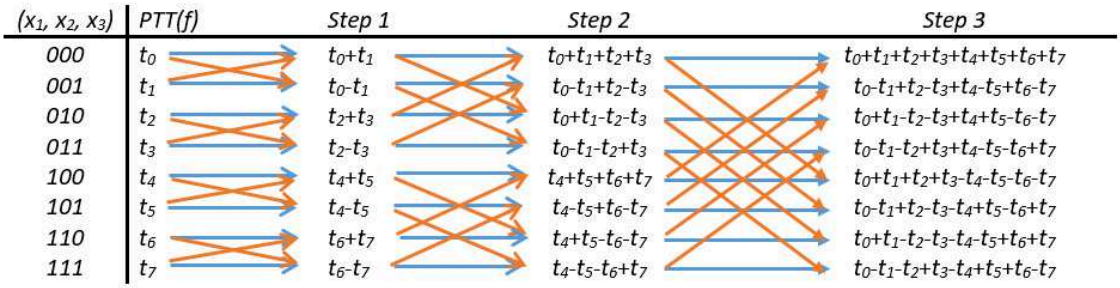
От дефиницията на H_n следва, че H_n е симетрична, редовете ѝ (както и стълбовете) образуват линейно пространство с размерност n, което в теорията на кодирането (без нулевата координата) е известно като симплексен код. Ако към това линейно пространство добавим съседния клас по вектора $(11\ldots1)$, получаваме код, известен като код на Reed-Muller от първи ред.

Нека $a = (a_1, a_2, \ldots, a_m)$ е двоичен вектор. Векторът $a^{(p)}$, който се получава от a след смяна на нулевите координати с 1, а единиците с -1, се нарича полярно представяне на a. Да разгледаме скаларното произведение $s = a^{(p)}.b^{(p)}$ над $\mathbb{Z}$. Ако s^- (съответно s^+) е броят на координатите, за които $a_j^{(p)}b_j^{(p)} = -1$ (съответно $a_j^{(p)}b_j^{(p)} = +1$), то $s^- = d(a,b)$ е броят на позициите, в които a и b се различават, а s^+ е броя на позициите в които a и b имат равни координати (броят на позициите, в които a и $11\ldots1 \oplus b$, допълнителния на b, се различават, или $s^- = d(a, b \oplus 11\ldots1)$). Имаме че $s = s^+ - s^-$ и $m = s^+ + s^-$ или $s^- = (m-s)/2,\ s^+ = (m+s)/2$.

Нека с $PTT(f)$ и H означим полярното представяне на вектора $TT(f)$ и матрицата H^+. Векторът $[W_f] = H \cdot (PTT(f))^t = (f^w(\overline{0}), f^w(\overline{1}), \ldots, f^w(\overline{2^n-1}))$, $W_f = (W_0, \ldots, W_{2^n-1})$, се нарича спектър на Walsh, а функцията $f^w(\overline{a})$ дефинира трансформация на Walsh. Компонентата W_i определя разстоянието на таблицата на истинност на функцията f до таблицата на истинност на линейната функция $x^{\oplus i}$, което е $(2^n - W_i)/2$ и до таблицата на истинност на афинната функция $1 + x^{\oplus i}$ което е $(2^n + W_i)/2$.

Аналогично на предния раздел и тук умножението $H \cdot PTT(f)^t$ на матрица по вектор може да се реализира като бътерфлай диаграма и съответен алгоритъм (Диаграма 2 (Фигура 2.2) и Алгоритъм 2.2). Този алгоритъм, както и първият, обхожда матрицата $S_{mat}^{(n)}$ на n стъпки, като започва от последния стълб. На j-тата стъпка според стойността на матрицата в i-тия ред на съответния стълб се прави промяна на стойността на $W_f[i]$ и $W_f[i+2^j]$. Този алгоритъм (както и първият) изцяло се определя от двоичното представяне на числата от 0 до $2^n - 1$. Въпреки че не е по-добър в порядък от досега известните, той е много по-компактен и много по-нагледен.

Нека го сравним с Алгоритъм 9.3 от класическата монография *Algorithmic Cryptanalysis*, стр. 275 [30]. Представеният алгоритъм има два пъти по-малко скаларни променливи. В Алгоритъм 9.3 [30] има 6 променливи от тип int, а в Алгоритъм 2.2 има само 3. Нашият алгоритъм има 3 присвоявания в най-вътрешния цикъл, а представеният в [30] има 5 присвоявания. В предложения алгоритъм вместо трите цикъла има два цикъла и един оператор IF. Условието в този условен оператор се проверява с две операции (както е показано в коментара). Аналогичен анализ може да се направи и на първия Алгоритъм 2.2.



Фигура 2.2: Диаграма 2: Бърза трансформация на Walsh

2.4 Коментари

Представеният подход е съвместна разработка с Илия Буюклиев и основно се базира на приложението на двоичното преставяне на целите неотрицателни числа, за описване на трансформациите и представянето на булевите функции и алгоритми свързани с тях. Алгоритъмът за преобразуване на булеви функции (от таблица на истинност в ANF) може да представлява част от алгоритъм за търсене на алгебрична степен, която е важна криптографска характеристика.

Algorithm 2.2 Fast Walsh Transform

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries
Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

$j \leftarrow 1$; $W_f \leftarrow PTT$;
while $(j < 2^n)$ do
  for $i = 0$ to $2^n - 1$ do
    if ((i & j)==0) then
      $temp \leftarrow W_f[i]$;
      $W_f[i] \leftarrow W_f[i] + W_f[i + j]$;
      $W_f[i + j] \leftarrow temp - W_f[i + j]$;
    end then
  end for
  $j \leftarrow 2 * j$;
end while.

Представеният алгоритъм за изчисление на спектъра на Walsh представлява част от последователната версия на алгоритъма за търсене на добри векторни булеви функции, конструирани от квазициклични кодове, за което ще говорим в петата глава. Основно алгоритмите са публикувани на конференцията на Съюза на математиците в България, проведена през 2015 г. [P2].

Глава 3

Алгоритми за трансформации на булеви функции и тяхна паралелна реализация на CUDA

Някои от най-важните криптографски свойства на булевите и векторните булеви (S-box) функции (нелинейност, автокорелация, диференциална еднаквост) са свързани с Walsh спектъра. Алгебричната степен също така, е важна криптографска характеристика на булевите и векторните булеви функции и тя е свързана с алгебричната нормална форма. В тази глава представяме паралелната имплементация на алгоритъма за изчисление на спектъра на Walsh (Fast Walsh Transform, FWT) и алгоритъм за изчисление на алгебричната нормална форма на булева функция с използване на Fast Möbius (Reed-Muller) Transform. Както при последователните алгоритми (Алгоритъм 2.2, Алгоритъм 2.1), така и тук използваме двоично представяне на цели неотрицателни числа при изпълнение на изчисленията. В предните глави дадохме основни понятия, свързани с булеви функции, спектъра на Walsh и алгебрична нормална форма. Тук са представени няколко алгоритми за изчисление на спектъра на Walsh. Първият е основен и най-лек за имплементация в CUDA. Даваме стъпка по стъпка изграждане и оптимизиране на алгоритъма за бързата трансформация на Walsh и всяка следваща версия е подобрение на предишната. Целта е да представим ефективен алгоритъм, да сравним оптимизационните стратегии и техники. От друга страна, даваме алгоритъм за изчисление на алгебричната нормална форма, като използваме оптимизационните техники, вече описани при алгоритъма за бързата трансформация на Walsh. Представени са получените експериментални резултати, отнасящи се до сравнение на последователната с паралелните имплементации на различните алгоритми и полученото ускорение.

В раздел 3.2 е представен различен подход за паралелна реализация на бързата трансформация на Walsh.

3.1 Общи положения на задачата за изчисление на спектъра на Walsh

Целта на тази част от главата е да се направи оценка на производителността на съвременните, сравнително евтини и общо използваеми NVIDIA GPUs при изчисление на Walsh спектъра на булева функция f. Трансформацията на Walsh както казахме има широк спектър на приложение и се използва в криптографията, обработка на сигнали, обработка на изображения, компресиране на данни и т.н. Тук ще представим няколко паралелни версии на алгоритъма, при които използваме основните концепции в Алгоритъм 2.2. За паралелна адаптация използваме CUDA C.

Досега са разработени последователни (CPU) реализации [30, 31], включени в библиотеки и математички софтуер, на пример Sage [58], Matlab [42], VBF Library [3], SET (S-box Evaluation Tool) [54] и т.н. За паралелна имплементация на трансформацията на Walsh се споменава при декодиране на low-density parity-check codes (LDPC кодове), [5, 15]. Също така, има няколко *GPU* библиотеки, които имплементират бътерфлай диаграми, cuFFT [45], BPLG [38], но не включват трансформацията на Walsh.

3.1.1 Паралелна реализация на FWT

Fast Walsh Transform може да се реализира паралелно с прилагане на основните концепции от Алгоритъм 2.2.

За описание на алгоритмите използваме следните обозначения за константи и променливи:

- $a\&b$, побитова операция AND за неотрицателните цели числа a и b;
- tID е индекса на текущата нишка в грида;
- bID е индекса на текущия блок;
- $tID\_inblock$ е индекса на нишката в текущия блок;
- $block\_size$ показва броя на нишките в блок;
- $block\_num$ е броя на блоковете в грида;
- $grid\_size$ е броя на всички нишки в грида, $grid\_size = block\_num * block\_size$;

Алгоритъм 3.1 е основан на последователния Алгоритъм 2.2, но с подходяща паралелна адаптация. Всички други алгоритми са модификация на

Algorithm 3.1 Parallel implementation of FWT

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries
Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

```
Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
        size ← 2^n;
        if (size <= 1024) then
           block_size ← size;
           block_num ← 1;
        end then
        else                    /* if size > 1024 */
           block_num ← size/1024;
           block_size ← 1024;
        end else

j ← 1; r ← 0; W_f ← PTT; temp ← 0;
while (j < size) do
        r ← r + 1;              /* r is the number of the current step */
         fwt_kernel(W_f, temp, r, j)      /*Launch kernel*/
        j ← 2 * j;              /* j is the size of the current step */
end while
if r is odd then W_f ← temp
Copy the result back to host
Cleanup memory
```

Алгоритъм 3.1.

Паралелната функция (**fwt_kernel**) в Алгоритъм 3.1 използва два масива, означени като W_f и $temp$, и две целочислени променливи r и j, където r означава текущата стъпка, а j е размера на текущата стъпка в бътерфлай алгоритъма. На всяка стъпка паралелната функция взема данни от един масив, в зависимост от стъпката, изчислява стойностите за съответната стъпка на бътерфлай алгоритъма и записва резултатите в другия масив. Използването на два масива се налага поради това, че няма глобална синхронизация между нишките от различните блокове на паралелната функция. Всъщност синхронизация постигаме с повторно извикване на паралелната функция от главната ($main$) последователна програма n на брой пъти.

fwt_kernel$(W_f, temp, r, j)$ Kernel, Algorithm 3.1

Input: The arrays W_f and $temp$ with 2^n entries, and the integers r and j.
Output: The arrays W_f and $temp$

```
i ← tID                    /* index of the thread */
if r is odd then           /* This determines where to save the intermediate result
*/
    if ((i&j) = 0) then
        value ← (W_f[i] + W_f[i + j]);
    end then
    else
        value ← (−W_f[i] + W_f[i − j]);
    end else
  temp[i] ← value;
end then
else
    if ((i&j) == 0) then
        value ← (temp[i] + temp[i + j]);
    end then
    else
        value ← (−temp[i] + temp[i − j]);
    end else
  W_f[i] ← value;
end else
```

Алгоритъм 3.1 използва само глобална памет, което води до загуба на ефективност. На практика необходимото време за извикване на паралелната функция (създаване и унищожаване на нишките) е незначително.

В случаите, когато имаме отклонение (divergence) на нишките в контейнер, може да се получи голяма загуба на ефикасността. Ето защо предлагаме Алгоритъм 3.2, който е модификация на Алгоритъм 3.1. Разликата е само в паралелната функция, където операторът $if-else$ е заменен с алгебричен израз. Изразът е:

$$value \leftarrow (1 - ii) * (W_f[i] + W_f[i + j]) + ii * (-W_f[i] + W_f[i - j]);$$

Променливата ii приема две стойности: $ii = 0$, ако $i\&j = 0$ и $ii = 1$ в противен случай.

Третият Алгоритъм 3.3 е модификация на Алгоритъм 3.1. В тази версия

fwt_kernel_no_if($W_f, temp, r, j$) Kernel, Algorithm 3.2

Input: The arrays W_f and $temp$ with 2^n entries, and the integers r and j.
Output: The arrays W_f and $temp$

```
i ← tID                    /* index of the thread */
ii ← (i&j)/j;              /* Conditional variable */
if r is odd then
    value ← (1 − ii) ∗ (W_f[i] + W_f[i + j]) + ii ∗ (−W_f[i] + W_f[i − j]);
    temp[i] ← value;
end then
else
    value ← (1 − ii) ∗ (temp[i] + temp[i + j]) + ii ∗ (−temp[i] + temp[i − j]);
    W_f[i] ← value;
end else
```

всяка нишка прави повече изчисления. Псевдокодът на паралелната имплементация на третата версия е показан в Алгоритъм 3.3.

Алгоритъм 3.3 има един и същ вход като Алгоритъм 3.1, но с допълнителен параметър M. В предните версии всяка нишка изчислява само една стойност за текуща стъпка в бътерфлай алгоритъма. В Алгоритъм 3.3 всяка нишка изчислява M стойности. Целта тук е да видим как това ще се отрази на производителността. Всъщност, само добавяме for цикъл в паралелната функция:

$$\text{for} \quad i = tID * M \quad \text{to} \quad (tID + 1) * M - 1 \quad \text{do.}$$

Освен това, конфигурацията на грида зависи от този параметър M. За изчисляване на спектъра на Walsh, голяма стойност на параметъра M означава по-малък брой нишки, но повече работа за нишка. Пример за конфигурация на грида е даден в псевдокода на Алгоритъм 3.3.

Четвъртият Алгоритъм 3.4 използва споделена памет в паралелната функция. На първия етап алгоритъмът използва споделена памет за изчисление на бързата трансформация на Walsh до определена стъпка (ограничението идва от това, че споделената памет се използва от нишките на един блок). Изчислението за следващите стъпки продължава в глобалната памет. Псевдокодът на паралелната имплементация на четвъртата версия е показан в Алгоритъм 3.4.

Алгоритъм 3.4 разполага с две паралелни функции. Първата използва споделена памет. Един блок подържа максимално работещи 1024 нишки (това е ограничение на графичната платка), които помежду си споделят данните. Ако $n > 10$, размерът на входния масив е по-голям от 1024. Затова входния

Algorithm 3.3 Parallel implementation of FWT

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries, and an integer $M = 2^s$, $s < n$;

Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads, the grid configuration depends on M
$size \leftarrow 2^n$;
if $(size/M <= 1024)$ then
    $block\_size \leftarrow size/M$;
    $block\_num \leftarrow 1$;
end then
else /* if size/M > 1024 */
    $block\_num \leftarrow (size/M)/1024$;
    $block\_size \leftarrow 1024$;
end else

$j \leftarrow 1$; $r \leftarrow 0$; $W_f \leftarrow PTT$
while $(j < 2^n)$ do
  $r \leftarrow r + 1$;
    fwt_kernel_M$(W_f, temp, r, j, M)$ /*Launch kernel*/
  $j \leftarrow 2 * j$;
end while
if r is odd then $W_f \leftarrow temp$
Copy the result back to host
Cleanup memory

fwt_kernel_M$(W_f, temp, r, j, M)$ Kernel, Algorithm 3.3

Input: The arrays W_f and $temp$ with 2^n entries, and the integers r, j, M.
Output: The arrays W_f and $temp$.

```
for i from tID * M to (tID + 1) * M − 1 do
  if r is odd then
      if ((i&j) == 0) then
          value ← (W_f[i] + W_f[i + j]);
      end then
      else
          value ← (−W_f[i] + W_f[i − j]);
      end else
    temp[i] ← value;
  end then
  else
      if ((i&j) == 0) then
          value ← (temp[i] + temp[i + j]);
      end then
      else
          value ← (−temp[i] + temp[i − j]);
      end else
    W_f[i] ← value;
  end else
end for
```

Algorithm 3.4 Parallel implementation of FWT

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries
Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

```
Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
        size ← 2^n
        if (size <= 1024) then
           block_size ← size;
           block_num ← 1;
        end then
        else                    /* if size > 1024 */
           block_num ← size/1024
           block_size ← 1024;
        end else

W_f ← PTT;
fwt_kernel_SM(W_f, block_size) /*Launch Shared memory, Kernel*/
j ← 1024; r ← 10;
while (j < 2^n) do
   r ← r + 1;
      fwt_kernel(W_f, temp, r, j)        /*Kernel, Algorithm 1*/
   j ← 2 * j;
end while
if r is odd then W_f ← temp
Copy the result back to host
Cleanup memory
```

fwt_kernel_SM($W_f, block\_size$) Kernel, Algorithm 3.4

Input: The array W_f with 2^n entries, and $block\_size$
Output: The array W_f.

```
Declare shared memory as the array tmpsdata of length block_size
Init tID, tID_inblock
i ← tID_inblock
value ← W_f[tID];        /*Local variable for every thread, taken from W_f*/
__syncthreads();
j ← 1;
while j < block_size do
  tmpsdata[i] ← value;
  __syncthreads();
    if ((i&j) == 0) then
        value ← (tmpsdata[i] + tmpsdata[i + j]);
    end then
    else
        value ← (−tmpsdata[i] + tmpsdata[i − j]);
    end else
  __syncthreads();
  j ← 2 * j
end while
W_f[tID] ← value.
```

масив с размер 2^n разделяме на части с размер от 1024 стойности. Всяка част се копира в споделената памет на съответния блок. Четенето и записването на данните е по-бързо в споделената памет. Освен това имаме механизъм за синхронизация. Това позволява тази функция да изпълнява фиксиран брой стъпки на алгоритъма (10 стъпки при максимално 1024 нишки за блок). При единайсетата стъпка нишките изискват данни извън тяхната споделена памет и се използва глобална памет за обмена на данни. Това се реализира във втората паралелна функция.

След извикване на първата паралелна функция, споделената памет е декларирана и в нея записваме данните от W_f. Всяка нишка взема две стойности от споделената памет, събира ги или ги изважда и съхранява резултата в регистър променлива. Следва синхронизация на нишките в блок и записване на резултата.

След приключване на първата паралелна функция, алгоритъмът извиква втората функция при нужда (ако $n > 10$). Втората паралелна функция е от

Algorithm 3.5. Parallel implementation of FWT

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries
Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

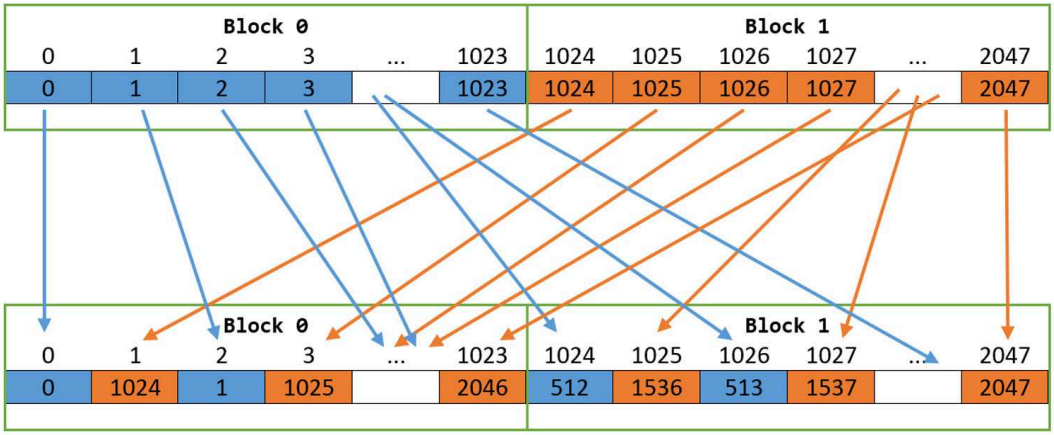
```
Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
    size ← 2^n
    if (size <= 1024) then
       block_size ← size;
       block_num ← 1;
    end then
    else                /* if size > 1024 */
       block_num ← size/1024
       block_size ← 1024;
    end else

W_f ← PTT
fwt_kernel_SM(W_f, block_size) /*Kernel, Algorithm 4*/
if (size > 1024) then              /* Shared memory + Memory pattern */
     fwt_kernel_SM_MP(W_f, block_num)
end then
Copy the result back to host
Cleanup memory
```

Алгоритъм 3.1 (паралелната функция **fwt_kernel**).

В Алгоритъм 3.5 използваме споделена памет, комбинирана с шаблона за пренареждане на паметта. Използваме този шаблон за подходящо пренареждане на междините резултати от стъпките на изчисление. Тук получаваме по-добри резултати в сравнение с другите версии. Тази версия е предназначена за вход с максимален размер до 2^{20} стойности или максималната дължина на вектора може да бъде 2^{20}. Псевдокодът на паралелната имплементация на петата версия е показан в Алгоритъм 3.5.

Алгоритъм 3.5 има две паралелни функции. Първата паралелна функция е **fwt_kernel_SM**, от Алгоритъм 3.4. Разликата между Алгоритъм 3.4 и Алгоритъм 3.5 е във втората паралелна функция. Тя е подобна на първата, но с добавен шаблон за пренареждане на паметта. След първата паралелна функция трябва да събираме стойностите $W_f[0]$ и $W_f[1024]$, $W_f[1]$ и $W_f[1025]$, и т.н..



Фигура 3.1: Шаблон за пренареждане на паметта за масива с размер от 2048 стойности

Те се намират в споделената памет на различни блокове.

За да можем да използваме отново споделената памет оптимално, трябва така да пренаредим данните, че да може да реализираме и следващите до 10 стъпки чрез споделена памет. Правим следното:

- Всяка нишка записва резултата, получен на 10-та стъпка в глобалния масив съответстващ на номера ѝ.

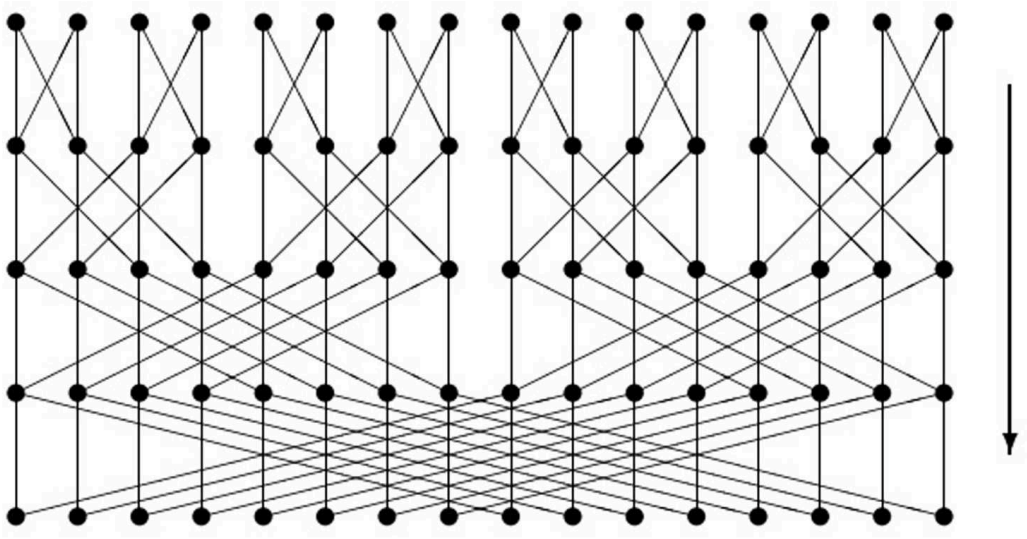
- Всяка нишка копира данните от клетката с номер, описан със следната формула:

$$ji = (tID\_inblock) * block\_num + bID.$$

След 10-та стъпка глобалния масив W_f получава резултата от изчисленията до момента, като j-та клетка получава резултат от j-та нишка. Да представим едномерен масив W_f като масив от буфери с големина $i = 2^n/1024$ и броят нишки за блок е 1024. Тези буфери образуват матрица M с 1024 реда и i стълба. Нека M^T е транспонираната на M. Тогава редовете на M^T, взети последователно, образуват масив W'_f. Този масив е необходимото за алгоритъма подреждане. На Фигура 3.1 ние показваме подреждане на стойностите във вектора W_f с размер от 2048 стойности преди 11-та стъпка на изчисление.

Шестият Алгоритъм 3.6 използва *warp shuffles*. *Warp shuffle* са машинни инструкции за NVIDIA графични процесори с изчислителна способност

```
for (int i=1; i<32; i*=2)
   value += __shfl_xor(value, i);
```



Фигура 3.2: Warp shuffles, $\_\_shfl\_xor$

3.0 и нагоре. Тези инструкции дават механизъм за предаване на данните (променливите в локалните регистри) между нишките от един контейнер без използване на друга памет. Има четири варианта на *warp shuffles*, от които за нашите нужди използваме $\_\_shfl\_xor(a, i)$, където a е локална регистър променлива, а i е цяло число, $0 < i \leq 16$ [20]. Тогава всички нишки в контейнера са разделени на 16 двойки (нишките с индекс $(0, i)$, $(1, i + 1), \ldots, (31 - i, 31)$). Стойността на променливата a на дадена нишка става достъпна за другата нишка в двойката (Фигура 3.2).

Алгоритъм 3.6 има две паралелни функции. Първата паралелна функция комбинира $\_\_shfl\_xor$ и споделена памет за изчисления до определена стъпка. От ограничението за споделената памет, комбинацията на $\_\_shfl\_xor$ и споделена памет може да изчислява до 10 стъпки от бързата трансформация Walsh (най-малко 5 стъпки). Първата паралелна функция е подобна на първата паралелна функция от Алгоритъм 3.4, с допълнение на $\_\_shfl\_xor$ като сегмент от кода, показан по-долу.

В този програмен сегмент правим бърза трансформация на Walsh на ниво контейнер. Инструкцията $\_\_shfl\_xor$ извършва основната работа тук.

Втората паралелна функция в Алгоритъм 3.6 е подобна на първата, но с

Algorithm 3.6. Parallel implementation of FWT

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries
Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
    $size \leftarrow 2^n$
    if ($size <= 1024$) then
        $block\_size \leftarrow size$;
        $block\_num \leftarrow 1$;
    end then
    else /* if size > 1024 */
        $block\_num \leftarrow size/1024$
        $block\_size \leftarrow 1024$;
        $for\_shfl \leftarrow block\_num$;
        if ($block\_num > 32$) then
            $for\_shfl \leftarrow 32$;
        end then
    end else

$W_f \leftarrow PTT$
fwt_kernel_shfl_xor_SM($W_f, block\_size$) /* Warp shuffles - Shared memory*/
if ($size > 1024$) then
    fwt_kernel_shfl_xor_SM_MP($W_f, block\_num, for\_shfl$)
    /* Warp shuffles - Shared memory - Memory pattern */
end then
Copy the result back to host
Cleanup memory

fwt_kernel_shfl_xor_SM$(W_f, block\_size)$ Kernel, Algorithm 3.6

Input: The array W_f with 2^n entries, and $block\_size$
Output: The array W_f.

Declare shared memory as the array $tmpsdata$ of length $block\_size$
Init tID, $tID\_inblock$
$i \leftarrow tID\_inblock$
$value \leftarrow W_f[tID]$; /*Local variable for every thread, taken from W_f*/
$\_\_syncthreads()$;
$j \leftarrow 1$
while $j < 32$ do
$ii \leftarrow (i\&j)/j$;
$value \leftarrow ii * (\_\_shfl\_xor(value, j) - value) + (1 - ii) * (\_\_shfl\_xor(value, j) + value)$;
$j \leftarrow 2j$
end while

while $j < block\_size$ do
$tmpsdata[i] \leftarrow value$;
$\_\_syncthreads()$;
if $((i\&j) == 0)$ then
$value \leftarrow (tmpsdata[i] + tmpsdata[i + j])$;
end then
else
$value \leftarrow (-tmpsdata[i] + tmpsdata[i - j])$;
end else
$\_\_syncthreads()$;
$j \leftarrow 2 * j$
end while
$W_f[tID] \leftarrow value$.

__shfl_xor. Сегмент от кода.

$j \leftarrow 1$
while $j < 32$ do
$ii \leftarrow (i\&j)/j$;
$value \leftarrow ii * (\_\_shfl\_xor(value, j) - value) + (1 - ii) * (\_\_shfl\_xor(value, j) + value)$;
$j \leftarrow 2 * j$
end while

fwt_kernel_shfl_xor_SM_MP$(W_f, block\_num, for\_shfl)$ Kernel, Algorithm 3.6

Input: The array W_f with 2^n entries, and $block\_size$
Output: The array W_f.

Declare shared memory as the array $tmpsdata$ of length $block\_size$
Init tID, $tID\_inblock$
$i \leftarrow tID\_inblock$
$ji = (tID\_inblock) * block\_num + bID$
$value \leftarrow W_f[ji]$; /*Local variable for every thread, taken from W_f*/
$\_\_syncthreads()$;
$j \leftarrow 1$
while $j < for\_shfl$ do
$ii \leftarrow (i\&j)/j$;
$value \leftarrow ii * (\_\_shfl\_xor(value, j) - value) + (1 - ii) * (\_\_shfl\_xor(value, j) + value)$;
$j \leftarrow 2j$
end while

while $j < block\_num$ do
$tmpsdata[i] \leftarrow value$;
$\_\_syncthreads()$;
if $((i\&j) == 0)$ then
$value \leftarrow (tmpsdata[i] + tmpsdata[i + j])$;
end then
else
$value \leftarrow (-tmpsdata[i] + tmpsdata[i - j])$;
end else
$\_\_syncthreads()$;
$j \leftarrow 2 * j$
end while
$W_f[ji] \leftarrow value$.

	Платформа 1	Платформа 2
CPU	Intel i3-3110M	Intel Xeon E5-2640
Memory	4 GB DDR3 1333 MHz	48GB DDR3 1333 MHz
OS	Win7 x64 SP1	Win7 x64 SP1
Compiler	MSVC 2010 SP1	MSVC 2012
GPU	GeForce GT 740M	GeForce GTX TITAN
Driver	v347.62, SDK 7.0	v347.62, SDK 7.0

Таблица 3.1: Описание на тест платформите

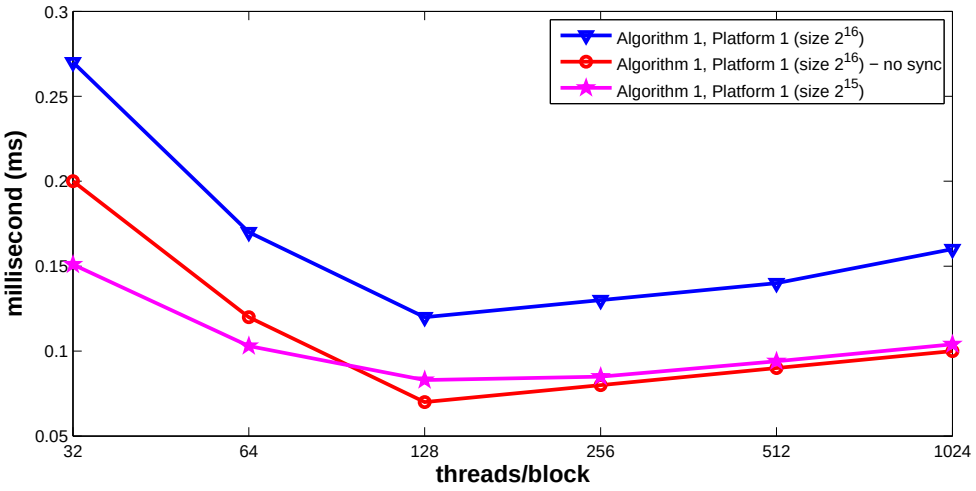
допълнителен шаблон за пренареждане на паметта. Вече споменахме, че с шаблона за пренареждане на паметта пренарежда данните в споделената памет по такъв начин, че стойностите от различни блокове са подредени за изпълнение на бързата трансформация на Walsh от началната стъпка. След определен брой стъпки отново пренареждаме (връщаме правилната подредба). Допълнителният параметър for_shfl използваме за проверка в конструкцията на $\_\_shfl\_xor$ в случай, че има по-малко от 5 стъпки на изчисления при втората паралелна функция.

В следващия подраздел даваме експериментални резултати и оценка на ефективността.

3.1.2 Експериментални резултати FWT

В тази част ще представим получените експериментални резултати. Тест - платформите, които използваме за експериментите, са дадени в Таблицата 3.1. Графичната карта на платформа 1, е NVIDIA GeForce GT 740M [49]. Тя има 384 процесора с тактова честота 0.9 GHz и честотна лента на паметта 28.8 GB/sec. Графична карта на платформа 2, е NVIDIA GeForce GTX TITAN [50]. Тя има 2688 процесора с тактова честота 837 MHz и честотна лента на паметта 288.4 GB/sec. Всички версии са имплементирани в паралелна изчислителна платформа и програмен модел CUDA [16]. При платформа 1 използваме CUDA Toolkit 7.0 и среда за разработване MS Visual Studio 2010. При платформа 2 използваме същата версия CUDA Toolkit 7.0, но среда за разработване MS Visual Studio 2012. Всички алгоритми изпълняваме в Active solution configuration - Release, а Active solution platform - Win32. Означаваме платформа 1 с $P1$, а платформа 2 с $P2$.

Програмите се изпълняват за размер на входа от 2^n елемента, където $n = 7, ..., 18$. Входния масив от данни се намира в GPU глобалната памет в началото на всеки експеримент, така че нямаме трансфер на данните между



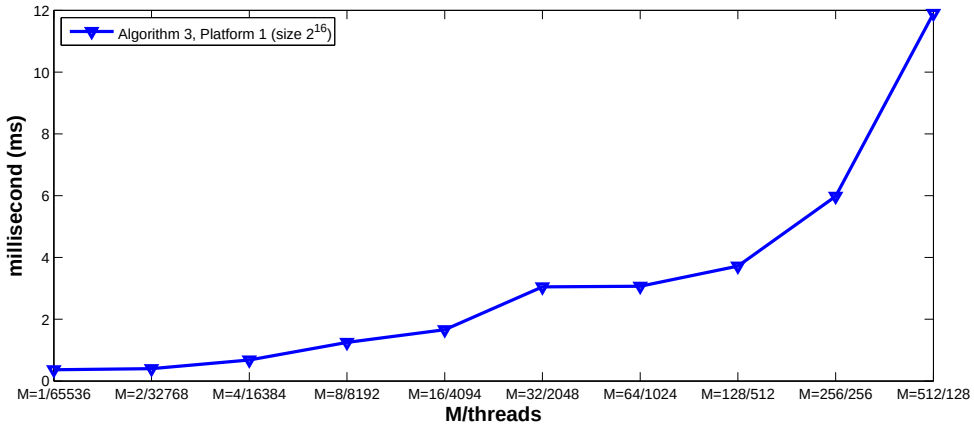
Фигура 3.3: Връзка между времето и броя нишки/блокове, (платформа 1)

глобалната памет на GPU и оперативната памет на CPU.

За да сравним с последователен алгоритъм, Алгоритъм 2.2 го реализираме в C++ език за програмиране с използване на MS Visual Studio 2010. Всички CPU примери са изпълнени на платформа 1 (Intel i3-3110M) в Active solution configuration - Release и Active solution platform - Win32.

Фигура 3.3 показва времето за изпълнение при изчисление на спектъра на Walsh за различен брой нишки за блок (Алгоритъм 3.1, платформа 1). Лините означени с ▼ и ● показват времето за изпълнение при изчисление на спектъра на Walsh за $n = 16$, като линията, означена с ● показва времето за изпълнение на програмата, когато няма синхронизация (когато нямаме синхронизация, получаваме неверни резултати и използваме тази програма за да представим как времето за синхронизация забавя изпълнението). Линията, означена с ★ показва времето на изпълнение при изчисление на спектъра на Walsh за $n = 15$ (в този случай спектърът е вектор с 2^{15} координати). Както се вижда от Фигура 3.3, може да заключим, че имаме оптимално време на изпълнение когато използваме 128 нишки в блок, а когато се увеличава броя нишки за блок се получава леко увеличаване на времето на изпълнение.

Резултатите при различни стойности на параметъра M спрямо времето на изпълнение (Алгоритъм 3.3, платформа 1) за $n = 16$ е показано на Фигура 3.4. Ако намалява броят нишки, изчисляваме повече Walsh коефициенти по нишка, а времето на изпълнение се увеличава. На Фигура 3.4, $M = 32/2048$ е представен грид, в който има 2048 нишки и всяка нишка изчислява $M = 32$



Фигура 3.4: Работа (M) на нишка срещу време на изпълнение, (платформа 1)

Walsh коефициенти.

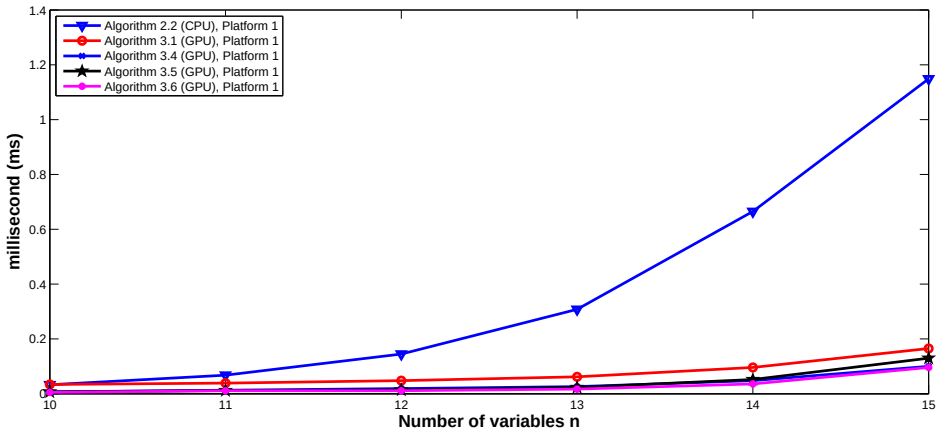
Сравнението между CPU Алгоритъм 2.2 и паралелните алгоритми (платформа 1) е показано на Фигура 3.5. Линията, означена с ▼ представлява C++ имплементацията на CPU Алгоритъм 2.2, а другите линии показват времето за изпълнение на Алгоритми 3.1, 3.4, 3.5, 3.6. Очевидно е, че в определена точка (за достатъчно голям размер) паралелната имплементация става по-бърза (подробни резултати за времето на изпълнение са посочени в Таблица 3.2).

Сравнението между Алгоритмите 3.1, 3.4, 3.5 и 3.6 е показано на Фигура 3.6 (платформа 1). Както се очаква, всяка по-оптимизирана версия има по-добро време за изпълнение от предишната за по-голямо n.

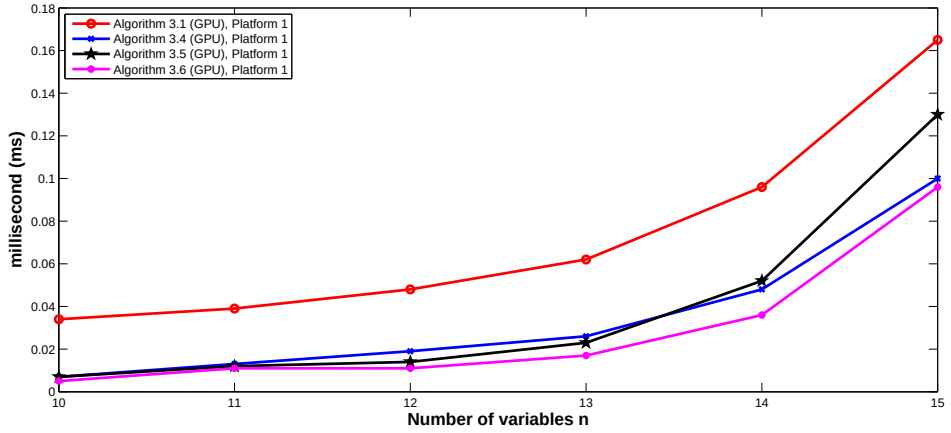
Една от основните цели е да се постигне ускорение на паралелната имплементация в сравнение с последователната имплементация. Формулата за ускорение е дефинирана в Глава 1 (формулата 1.4).

Таблица 3.2 показва времето за изпълнение (в ms) на CPU реализацията и реализацията на паралелните алгоритми за различни размери на входните данни, както и ускорението между различните имплементации. Ускорението на паралелните алгоритми с номер i означаваме с $S_i, i = 1, ..., 6$. От таблицата се вижда, че CPU имплементацията има по-добро време за малки стойности на n. При паралелната имплементация, когато n се увеличава, имаме повече нишки и изчислението е по-бързо в сравнение с последователната имплементация.

Експериментални резултати от паралелните имплементации на платформа 2 за Алгоритми 3.1, 3.4, 3.5, 3.6, са показани на Фигура 3.7. Представени са експерименталните резултати за стойности на $n = 14, ..., 18$.



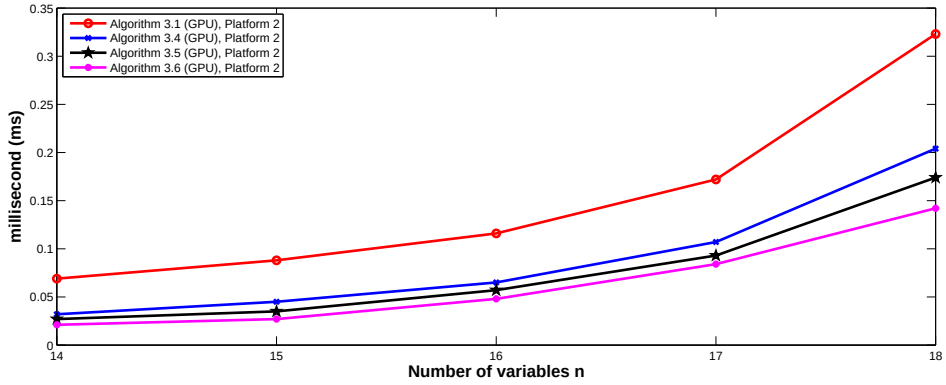
Фигура 3.5: Време на изчисление $[W_f]$ CPU срещу различните GPU имплементации, (платформа 1)



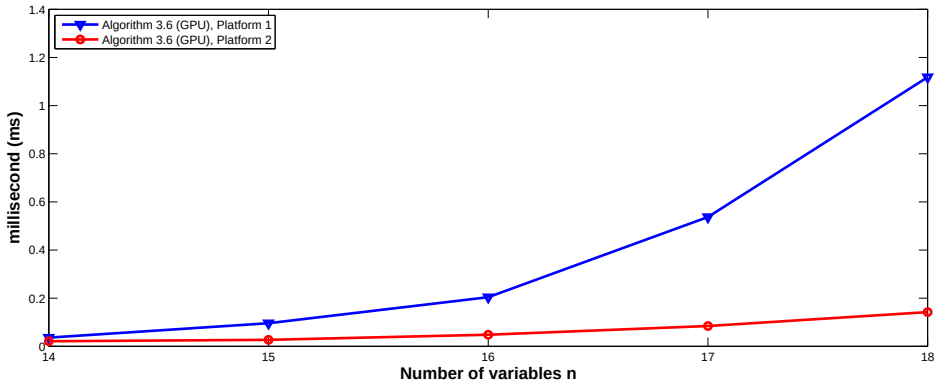
Фигура 3.6: Време на изчисление $[W_f]$, GPU имплементации, (платформа 1)

#comp. words	CPU (ms)	A1 (ms)	S_1	A4 (ms)	S_4	A5 (ms)	S_5	A6 (ms)	S_6
2^7	0,003	0,024	< 1	0,0066	< 1	0,0066	< 1	0,0056	< 1
2^8	0,007	0,026	< 1	0,0066	1.060	0,0066	1.060	0,0057	1.222
2^9	0,015	0,028	< 1	0,0069	2.272	0,0069	2.272	0,0058	2.547
2^{10}	0,033	0,034	< 1	0,0071	4.647	0,0071	4.647	0,0059	5,584
2^{11}	0,068	0,039	2	0,013	5.230	0,0124	5.483	0,0116	5.862
2^{12}	0,145	0,048	3,02	0,019	7.631	0,0147	9.863	0,0119	12.156
2^{13}	0,308	0,062	4,967	0,026	11.84	0,023	13.39	0,0174	17.647
2^{14}	0,665	0,096	6,927	0,048	13.85	0,052	12.78	0,0366	18.085
2^{15}	1,148	0,165	6,961	0,1	11.48	0,13	8.836	0,0965	11.901
2^{16}	3,116	0,366	8,513	0,24	12.98	0,28	11.12	0,2042	15.259
2^{17}	6,87	1,561	4,401	0,85	8.082	0,595	11.54	0,5379	12.771
2^{18}	14,81	3,571	4,149	2,058	7.207	1,207	12.27	1.1187	13.245

Таблица 3.2: FWT: CPU срещу GPU имплементации, платформа 1



Фигура 3.7: Време на изчисление $[W_f]$, GPU имплементации, (платформа 2)



Фигура 3.8: Време на изчисление (A6): платформа 1 срещу платформа 2

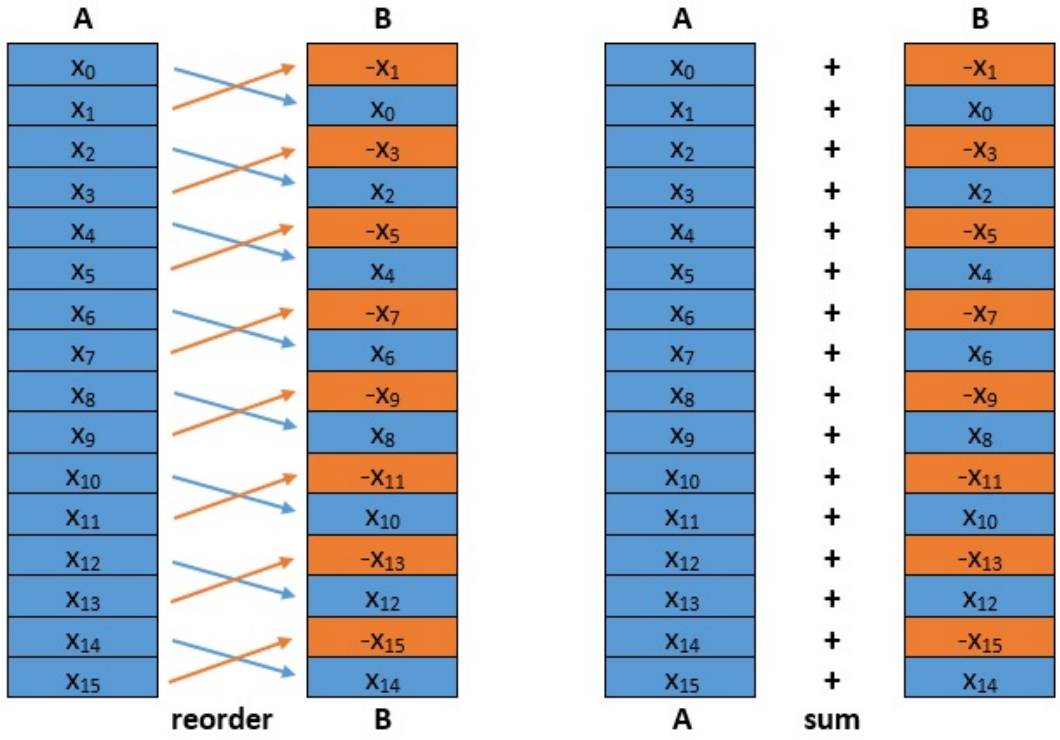
#comp.words	A6,P1(ms)	A6,P2(ms)	S_p(A6):P1vs.P2	CPU(ms)	S_p:P2(A6)vs.CPU
2^{14}	0,036	0,021	1.71x	0,665	31.66x
2^{15}	0,096	0,027	3.55x	1,148	42.51x
2^{16}	0,204	0,048	4.25x	3,116	64.91x
2^{17}	0,537	0,084	6.40x	6,87	81.78x
2^{18}	1.118	0,142	7.87x	14,81	104.30x

Таблица 3.3: FWT: Алгоритъм 6: платформа 1 срещу платформа 2 и CPU срещу платформа 2(A6)

На Фигура 3.8 е показано сравнение между най-добрия Алгоритъм 3.6 за платформа 1 и платформа 2. Както виждаме на Фигура 3.8, разликата във времето на изпълнение между платформите се увеличава, когато се увеличава размерът на данните. В Таблица 3.3 е дадено подробно сравнение между платформите за Алгоритъм 3.6 и сравнение между CPU реализацията и Алгоритъм 3.6, платформа 2. Значителното ускорение, което се получава при платформа 2, се дължи на факта, че тази платформа има по-добри хардуерни характеристики.

3.2 Компактна версия на паралелна реализация на FWT

Идеята на алгоритъма за този подход при реализация на бързата трансформация на Walsh има различна концепция от горе представената. Най-лесният



Фигура 3.9: Една стъпка на изпълнение (първата стъпка), fwt_kernel_KV

начин за обяснение на идеята е с помощта на два вектора, които се събират (Фигура 3.9). При реализацията на този подход имаме два етапа при една стъпка на изчисление на бързата трансформация на Walsh. Първият етап (Фигура 3.9, reorder) има шаблон за копиране и пренареждане на стойностите от вектора A във вектора B. Всъщност, подредбата на стойностите във вектора B е направена така, че за втория етап (Фигура 3.9, sum) събирането на векторите A и B образува една стъпка на бързата трансформация на Walsh.

Псевдокодът на паралелната функция на този подход е представен в **fwt_kernel_KV**.

Паралелната функция **fwt_kernel_KV** има за вход вектор W_f, а за изход - резултатът от изчисленията (вектор $W_f result$). Тя използва споделена памет и локални регистри за изчисленията. Типът на използваната памет позволява паралелната функция да изпълнява определен брой стъпки на бързата трансформация на Walsh (който е 10 стъпки при максимално 1024 нишки за блок).

Реализацията на целия алгоритъм за бързата трансформация на Walsh

fwt_kernel_KV($W_f, W_f result, block\_size$) Kernel

Input: The array W_f with 2^n entries, and $block\_size$
Output: The array $W_f result$.

Declare shared memory as the array $tmpsdata$ of length $block\_size$
Init tID, $tID\_inblock$
$i \leftarrow tID\_inblock$
$val, f1, r \leftarrow 1$, $j \leftarrow 1$; //declare local variable
$value \leftarrow W_f[tID]$; /*Local variable for every thread, taken from W_f*/
$\_\_syncthreads()$;

while $j < block\_size$ do
  $tmpsdata[i] = value$;
  $\_\_syncthreads()$;
    $f1 \leftarrow 1 - 2 * (tid \gg (r-1)\&1)$;
    $val \leftarrow tmpsdata[tid + f1 * j] * (-f1)$;
    $value \leftarrow value + val$;
    $j \leftarrow 2 * j$;
    $r++$;
  $\_\_syncthreads()$;
end while
$W_f result[tID] \leftarrow value$;

е същата, както в горе представените алгоритми, които съдържат две паралелни функции. При втората функция, също така, имаме нужда от шаблон за пренареждане на паметта. Няма да навлизаме повече в подробности и да представяме втора паралелна функция, защото идеята на реализацията виждаме от представените по-горе версии на алгоритъма.

Този подход на реализация дава резултати, близки до най-добрите по-горе описани версии и поради това не е направен подробен анализ и сравнение. Недостатъкът на подхода е, че резултатът от изчислението на спектъра на Walsh е в обратна подредба.

3.3 Общи положения на задачата за изчисление на алгебрична нормална форма

Целта на тази част от главата е да се направи оценка на производителността на съвременните, сравнително евтини и общо използваеми NVIDIA GPUs при изчисление на алгебричната нормална форма (ANF) на булева функция f като знаем таблицата на истинност (TT) и обратно. Булевите функции са представени чрез таблица на истинност, а за да изчислим тяхната алгебрична степен се нуждаем от алгебричната им нормална форма. Ефективен подход за това изчисление дава Möbius (Reed-Muller) Transform. Тук представяме побитова паралелна реализация на алгоритъм за изчисление на алгебричната нормална форма (ANF) на булева функция чрез двоичната Möbius (Reed-Muller) Transform. Чрез побитовото представяне постигаме две цели: бързо изчисление и лесен трансфер и манипулация с данните. Ползата от тази реализация е ефективно и бързо изчисление при булева функция с много променливи, както и векторна булева функция с голям размер. Нашият алгоритъм е предназначен за NVIDIA GPUs с изчислителна способност 3.0 и по-висока [16].

Основната идея на нашата реализация е свързана със следните факти: (1) използваме две нива на компютърен паралелизъм - първият е побитовото представяне, а вторият е използването на хиляди нишки; (2) комбинираме побитовите операции, warp shuffles инструкциите, споделената памет и шаблон за пренареждане на паметта; (3) всички константи (или маски), които използваме при побитовото изчисление, са фиксирани преди компилацията.

До сега са известни някои алгоритми за изчисление на алгебрична нормална форма, които са свързани с Möbius (Reed-Muller) Transform [6, 30, 31]. Също така, има много последователни (CPU) реализации, инструменти, библиотеки и математически софтуер (виж в раздел 3.1).

Както вече споменахме, за изчисление на алгебрична нормална форма на

булева функция може да използваме FMT, която е зададена чрез бътерфлай диаграма и съответен алгоритъм със сложност $O(n2^n)$ (Фигура 2.1, [30]). В [P2] (Глава 2), представихме Алгоритъм 2.1, който използва двоичното представяне на целите неотрицателни числа.

Преди да представим нашия следващ алгоритъм, нека да разгледаме следните вектори с дължина 2^n:

$$v_0 = (1010\ldots10), v_1 = (11001100\ldots1100), \ldots, v_{n-1} =$$

$$= (\underbrace{11\ldots1}_{2^{n-1}}\underbrace{00\ldots0}_{2^{n-1}}).$$

Всъщност векторът v_i се състои от 2^{n-1-i} екземпляри от $\underbrace{11\ldots1}_{2^i}\underbrace{00\ldots0}_{2^i}$, $i = 0, \ldots, n-1$.

Нека shr_t, е вектор, който е циклично преместен вдясно на t позиции. Тогава алгоритъмът FMT, представен в диаграмата, може да се запише по следния начин:

$$\begin{aligned} &for \;\; i = 0 \; to \; n-1 \; do \\ &\quad f = f \oplus (shr_{2^i}(v_i) \& f), \end{aligned}$$

където f е трансформирания вектор.

За да реализираме FMT използваме побитово представяне на векторите $TT = (f_0, f_1, \ldots, f_{2^n-1})$ в масив, на който координатите са 64-битови цели неотрицателни числа (C++ тип на променлива *unsigned long long*). През първите шест стъпки на алгоритъма използваме маски, които са векторите $v_0, \ldots, v_5$ с дължина от 64. Тъй като компютърната дума в нашия случай съдържа 64 бита, след шестата стъпка не можем да използваме маски. Следователно Алгоритъм 3.7 има две части. Първата изчислява шест стъпки, а втората е за следващите стъпки.

Алгоритъм 3.7 е комбинация на побитовия FMT алгоритъм представен в [6, 30] (до определено ниво) и FMT алгоритъма, представен в [P2]. Тук също използваме *unsigned long long* тип данни (с размер от 8 байта). Лесно можем да променим алгоритъма за тип данни с по-малък размер. Fast Möbius (Reed-Muller) Transform може да се реализира паралелно с прилагане на основните концепции от Алгоритъм 3.7. За паралелна адаптация използваме *CUDA C*, така че нашия алгоритъм използва различни оптимизационни техники, модели и типове на памет, за да получим по-добра производителност и ефективност.

Algorithm 3.7. (bit-wise) Fast Möbius Transform

Input: The Truth Table $int64TT$ of the Boolean function f, whose coordinates are 64-bit words. The number of coordinates is $NumInt = 2^n/64$.

Output: The Algebraic Normal Form $int64ANF$ of the Boolean function f, whose coordinates are 64-bit words

/*bitwise Fast Möbius Transform */
$ANF \leftarrow int64TT$; $NextInt \leftarrow 0$;
while $(NextInt < NumInt)$ do
$value \leftarrow ANF[NextInt]$;
/*First six steps (with a mask)*/
$value \oplus = (value \& 12297829382473034410) >> 1$;
$value \oplus = (value \& 14757395258967641292) >> 2$;
$value \oplus = (value \& 17361641481138401520) >> 4$;
$value \oplus = (value \& 18374966859414961920) >> 8$;
$value \oplus = (value \& 18446462603027742720) >> 16$;
$value \oplus = (value \& 18446744069414584320) >> 32$;
$NextInt \leftarrow NextInt + 1$;
$ANF[NextInt] \leftarrow value$;
end while

/*Next steps (without a mask)*/
$j \leftarrow 1$;
while $(j < NumInt)$ do
  for $i = 0$ to $NumInt - 1$ do
    if $((i \& j) == j)$ then
      $ANF[i] \leftarrow ANF[i] \oplus ANF[i - j]$;
        end then
  end for
  $j \leftarrow 2 * j$;
end while

3.3.1 Паралелна реализация на FMT

В този подраздел ще представим паралелната реализация на Reed-Muller Transform, имплементиран в $CUDA\ C$.

Алгоритъмът 3.8 е основан на последователния Алгоритъм 3.7, но с подходяща паралелна адаптация. Паралелните функции на Алгоритъм 3.8 използват масив, означен с ANF, както и целочислени стойности $block\_size$, $block\_num$ и for_shfl. Самият алгоритъм работи за масиви с размер от $2^{11}/64$ до $2^{26}/64$ компютърни думи.

Паралелната реализация се влияе напълно от бързия достъп до данните от всяка нишка, както и бързият обмен на данните между нишките, което значително зависи от типа на използваната памет (подраздел 1.4.1), както и от това дали използваме warp shuffles инструкциите.

Структурата на грида изграждаме така, че всички нишки, в нашия случай техният брой е $2^n/64$ (което е дължината на входния масив), са разположени в блокове с максимален размер от 1024 нишки (зависи от платформата, която използваме).

За описание на псевдокода на алгоритмите използваме същите означения като в подраздел 3.1.1. Добавяме и означение за XOR операция:

- $a \oplus b$ - побитова операция XOR за неотрицателните цели числа a и b;

Нашият алгоритъм съдържа няколко елемента, които образуват две паралелни функции. Алгоритъм 3.8 работи по следния начин:

1. В началото заделяме памет, записваме входните данни от хоста в устройството и конфигурираме грида от нишки и блокове.

2. На всяка нишка съпоставяме съответен елемент от входния масив и след това изпълняваме шест стъпки с използване на маски. Първата част е същата като при последователния Алгоритъм 3.7.

3. За следващите пет стъпки използваме wrap shuffles (инструкции за обмен на данните между нишките от един контейнер). Тази част е показана в код сегмента $\_\_shfl\_up$. Ние използваме инструкцията $\_\_shfl\_up(a, i)$ където a е локална променлива, а i е целочислена променлива [20]. При тази инструкция всяка нишка има достъп до локалната променлива на нишка с номер по-голям с i от текущата от същия контейнер.

4. На следващата стъпка данните се намират в локалната памет на нишките в различни контейнери, но в един блок. Следователно използваме споделена памет за тяхнато подходящо пренареждане. Този подход се прилага до 16-та стъпка.

Algorithm 3.8 Parallel implementation of FMT

Input: The Truth Table $int64TT$ of the Boolean function f, whose coordinates are 64-bit words. The number of coordinates is $NumInt = 2^n/64$.

Output: The Algebraic Normal Form $int64ANF$ of the Boolean function f, whose coordinates are 64-bit words

```
Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
    size ← NumInt;
    if (size <= 1024) then
       block_size ← size;
       block_num ← 1;
    end then
    else                    /* if size > 1024 */
       block_num ← size/1024;
       block_size ← 1024;
       for_shfl ← block_num;
       if (block_num > 32) then
            for_shfl ← 32;
       end then
    end else

ANF ← int64TT;
/*Kernel: bitwise - Warp shuffles - Shared memory*/
bitwise_fmt_kernel_shfl_up_SM(ANF, block_size)
if (size > 1024) then
/*Kernel: Warp shuffles - Shared memory - Memory pattern */
    fmt_kernel_shfl_up_SM_MP(ANF, block_num, for_shfl)
end then
Copy the result back to host
Cleanup memory
```

__shfl_up. Wrap shuffles код сегмент.

$j \leftarrow 1$; $f1 \leftarrow 1$; $r \leftarrow 1$;
while $j < 32$ do
$f1 \leftarrow (tid >> (r-1)\&1)$;
$value \leftarrow (value \oplus \_\_shfl\_up(value, j)) * (f1) + value * (1 - f1)$;
$r \leftarrow r + 1$;
$j \leftarrow 2 * j$;
end while

5. Предходните елементи формират паралелна функция (Kernel 1), именувана като **bitwise_fmt_kernel_shfl_up_SM**.

   При следващите стъпки нишките използват данни, които са в различни блокове, заради което прилагаме шаблон за пренареждане на паметта. Подробно описание на шаблона за пренареждане на паметта е дадено в подраздел 3.1.1, в частта за описанието на Алгоритъм 3.5. Шаблонът за пренареждане на паметта пренарежда споделената памет по такъв начин, че стойностите от различни блокове са подредени за изпълнение на FMT от началната стъпка.

   След пренареждането стъпките $7-16$ могат да се повторят отново. Това реализираме чрез втора паралелна функция (Kernel 2), именувана като **fmt_kernel_shfl_up_SM_MP**.

В следващия подраздел даваме експериментални резултати и оценка на ефективността.

3.3.2 Експериментални резултати FMT

В тази част ще представим получените експериментални резултати. Тест - платформите, които използваме за експериментите, са дадени в Таблицата 3.1. Единствената разлика е версията на CUDA Toolkit, която е 8.0 и версията на драйверите $v378.92$. Настройките на средата за разработка на платформите са вече описани в подраздел 3.1.2. Означаваме платформа 1 с $P1$, а платформа 2 с $P2$.

Програмите се изпълняват за размер на входа от $2^n/64$ елемента, където $n = 11, ..., 26$. Входният масив от данни се намира в GPU глобалната памет в началото на всеки експеримент, така че нямаме трансфер на данните между глобалната памет на GPU и оперативната памет на CPU.

За да сравним с последователен алгоритъм, Алгоритъм 3.7 се реализира на езика за програмиране C++ с използване на MS Visual Studio 2010.

bitwise_fwt_kernel_shfl_up_SM$(ANF, block\_size)$ Kernel 1

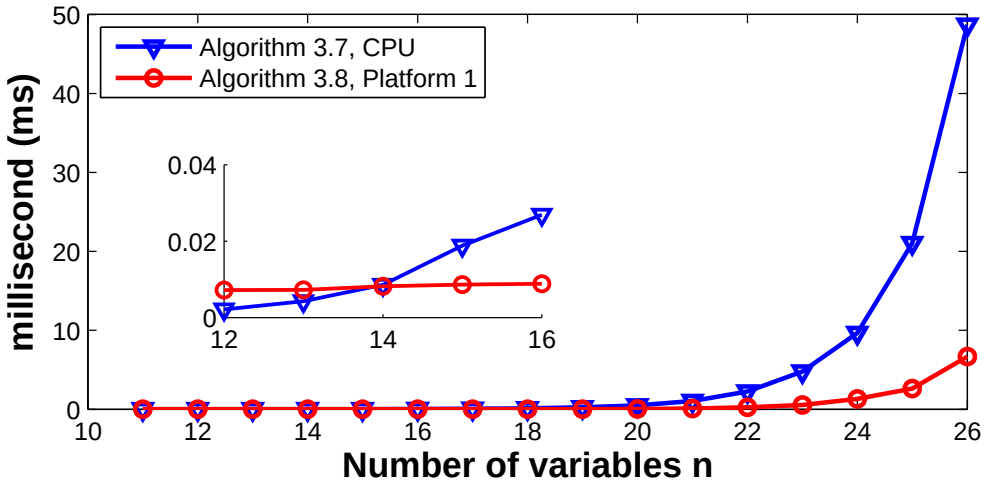
Input: The array ANF with $NumInt = 2^n/64$ entries, and $block\_size$
Output: The array ANF

Declare shared memory as the array $tmpsdata$ of length $block\_size$
Init tID, $tID\_inblock$;
$i \leftarrow tID\_inblock$;
/*Local variable for every thread, taken from ANF*/
$value \leftarrow ANF[tID]$;
/*Six steps of bitwise, Fast Möbius Transform */
$value \oplus = (value \& 12297829382473034410) >> 1$;
$value \oplus = (value \& 14757395258967641292) >> 2$;
$value \oplus = (value \& 17361641481138401520) >> 4$;
$value \oplus = (value \& 18374966859414961920) >> 8$;
$value \oplus = (value \& 18446462603027742720) >> 16$;
$value \oplus = (value \& 18446744069414584320) >> 32$;
/*Five steps of Warp shuffles, Fast Möbius Transform */
$j \leftarrow 1$; $f1 \leftarrow 1$; $r \leftarrow 1$;
while $j < 32$ do
$f1 \leftarrow (tid >> (r-1) \& 1)$;
$value \leftarrow (value \oplus \_\_shfl\_up(value, j)) * (f1) + value * (1 - f1)$;
  $r \leftarrow r + 1$;
  $j \leftarrow 2 * j$;
end while
/*Five steps of Shared memory, Fast Möbius Transform */
while $j < block\_size$ do
  $tmpsdata[i] \leftarrow value$;
  $\_\_syncthreads()$;
    if $((i \& j) == j)$ then
      $value \leftarrow value \oplus tmpsdata[i - j]$;
    end then
  $j \leftarrow 2 * j$;
end while
$ANF[tID] \leftarrow value$;

fmt_kernel_shfl_up_SM_MP$(W_f, block\_num, for\_shfl)$ Kernel 2

Input: The array ANF with $NumInt = 2^n/64$ entries, and $block\_size$
Output: The array ANF

Declare shared memory as the array $tmpsdata$ of length $block\_size$
Init tID, $tID\_inblock$;
$i \leftarrow tID\_inblock$;
$ji = (tID\_inblock) * block\_num + bID$;
/*Local variable for every thread, taken from ANF*/
$value \leftarrow ANF[ji]$;
/*Five steps of Warp shuffles */
$j \leftarrow 1$; $f1 \leftarrow 1$; $r \leftarrow 1$;
while $j < for\_shfl$ do
$f1 \leftarrow (tid >> (r-1)\&1)$;
$value \leftarrow (value \oplus \_\_shfl\_up(value, j)) * (f1) + value * (1 - f1)$;
  $r \leftarrow r + 1$;
  $j \leftarrow 2 * j$;
end while
/*At most five steps of Shared memory */
while $j < block\_num$ do
  $tmpsdata[i] \leftarrow value$;
  $\_\_syncthreads()$;
    if $((i\&j) == j)$ then
        $value \leftarrow value \oplus tmpsdata[i - j]$;
    end then
  $j \leftarrow 2 * j$;
end while
$ANF[ji] \leftarrow value$;



Фигура 3.10: Алгоритъм 3.7, CPU срещу Алгоритъм 3.8, платформа 1

Всички CPU примери са изпълнени на платформа 1 (Intel i3-3110M поради това, че няма значителна разлика във времето на изпълнение между платформите $\pm 5\%$) в Active solution configuration - Release и Active solution platform - Win32.

Сравнението между CPU Алгоритъм 3.7 и паралелния Алгоритъм 3.8 (платформа 1) е показано на Фигура 3.10. Линията, означена с ▼ представлява времето за изпълнение на CPU Алгоритъм 3.7, имплементацията на C++, а линията, означена с ● показва времето за изпълнение на Алгоритъм 3.8. Очевидно е, че в определена точка (за достатъчно голям размер) паралелната имплементация става по-бърза (подробни резултати от времето на изпълнение са посочени в Таблица 3.4).

Една от основните цели е да се оцени ускорението на паралелната имплементация по отношение на последователната имплементация. Формулата за ускорение е дефинирана в Глава 1 (формулата 1.4).

Таблица 3.4 показва времето за изпълнение (в ms) на CPU реализацията и реализацията на паралелния алгоритъм за различни размери на входните данни, както и ускорението между GPU и CPU реализацията. От таблицата се вижда, че CPU имплементацията има по-добро време за малки стойности на n. При паралелната имплементация, когато n се увеличава, имаме повече нишки и изчислението е по-бързо в сравнение с последователната имплементация.

Експериментални резултати от паралелната имплементация на платформа 2 за Алгоритъм 3.8, са показани на Фигура 3.11. Включени са експеримен-

#comp. words	CPU (ms)	P1, GPU (ms)	S_p: P1 vs. CPU
$2^{11}/64$	0.0020	0.007136	/
$2^{12}/64$	0.0022	0.007168	/
$2^{13}/64$	0.0043	0.007264	/
$2^{14}/64$	0.0086	0.008160	1.06x
$2^{15}/64$	0.0188	0.008648	2.18x
$2^{16}/64$	0.0269	0.008846	3.06x
$2^{17}/64$	0.0629	0.011456	5.72x
$2^{18}/64$	0.1163	0.014688	8.31x
$2^{19}/64$	0.2634	0.023584	11.21x
$2^{20}/64$	0.5050	0.041087	12.32x
$2^{21}/64$	1.0601	0.104358	10.2x
$2^{22}/64$	2.2532	0.253005	9.01x
$2^{23}/64$	4.7925	0.542837	8.9x
$2^{24}/64$	9.6577	1.329511	7.32x
$2^{25}/64$	21.041	2.632896	8.0x
$2^{26}/64$	48.706	6.689343	7.3x

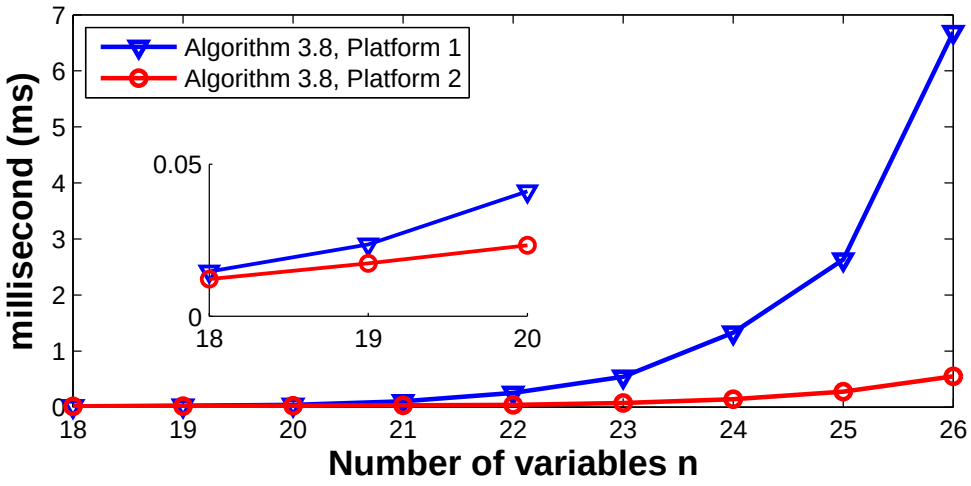
Таблица 3.4: FMT: CPU срещу GPU имплементации, платформа 1

талните резултати за размер на входния вектор 2^n, $n = 18, ..., 26$.

На Фигура 3.11 е показано сравнение на Алгоритъм 3.8 за платформа 1 и платформа 2. Както виждаме на Фигура 3.11 разликата във времето на изпълнение между платформите се увеличава, както се увеличава размерът на данните. В Таблица 3.5 е дадено подробно сравнение между платформите за Алгоритъм 3.8 и сравнение между CPU реализацията и Алгоритъм 3.8, платформа 2. Както се вижда от Таблица 3.5, за размер $2^{26}/64$ има ускорение от $90x$ пъти на Алгоритъм 3.8, платформа 2 в сравнение с CPU. Значителното ускорение, което се получава при платформа 2 се дължи на факта, че тази платформа има по-добри хардуерни характеристики.

3.4 Коментари

В тази глава е представен паралелен модел за пресмятане на спектъра на Walsh и алгебричната нормална форма на булева функция с използване на NVIDIA графичните процесори, като е направено сравнение с последователно реализирани алгоритми. Това е един пример, където паралелното изчисление е ефективно и можем да видим ползите от него. Тук показваме стъпка по стъпка



Фигура 3.11: Време на изчисление (Алгоритъм 1): платформа 1 срещу платформа 2

#comp. words	P1 (ms)	P2 (ms)	S_p: P1 vs P2	CPU (ms)	S_p: CPU vs P2
$2^{18}/64$	0.014	0.012	1.16x	0.116	9.66x
$2^{19}/64$	0.023	0.017	1.35x	0.263	15.47x
$2^{20}/64$	0.041	0.023	1.78x	0.505	22x
$2^{21}/64$	0.104	0.030	3.46x	1.060	35.33x
$2^{22}/64$	0.253	0.040	5.87x	2.253	56.32x
$2^{23}/64$	0.542	0.074	7.32x	4.792	64.86x
$2^{24}/64$	1.329	0.141	9.42x	9.657	68.49x
$2^{25}/64$	2.263	0.274	8.25x	21.041	78x
$2^{26}/64$	6.689	0.549	12.18x	48.706	90.2x

Таблица 3.5: FMT сравнение: платформа 1 (Алгоритъм 3.8) срещу платформа 2 (Алгоритъм 3.8) и CPU (Алгоритъм 3.8)

подобряването и оптимизацията на основния алгоритъм, което увеличава производителността. Изборът на правилна оптимизационна техника и подходящи методи осигурява увеличаване на ефективността и производителността.

Важно е да споменем, че при алгоритмите, които използват шаблон за пренареждане на паметта (това важи за FWT при $11 \leq n \leq 20$), доколкото не е от съществено значение правилното пренареждане на изчислените Walsh коефициенти, тази стъпка може да се пропусне. Вместо пренареждане на изчислените Walsh коефициенти, нишките записват изчислените стойностите директно в глобалната памет според индекса. Самото пропускане на стъпката за пренареждането на изчислените коефициенти спестява ресурс, от друга страна ускорява изпълнението с 10-20%.

Разработените алгоритми в тази глава са съвместна работа с Илия Буюклиев. Част от алгоритмите са приети за публикуване или се редактират [P8], [P7]. Някои идеи и предварителни версии на алгоритмите са докладвани на [D2], [D4], [D5]. Също така, някои идеи и версии на алгоритъма за изчисление на спектъра на Walsh са публикувани в [P4], [P5].

В публикация [P3], докладвана на [D3], е представена паралелна реализация на изчисление на спектъра на Walsh, чрез паралелно изчисление на вектор $(PTT(f))$ с матрица (H_n). Тази реализация, в сравнение с останалите паралелни реализации на FWT, има по-голяма сложност и изисква повече памет и следователно доста повече време за изпълнение. Поради тази причина тук не е представена тази публикация.

В раздел 3.2 е представен изцяло различен подход за реализация на бързата трансформация на Walsh. Алгоритъмът в този раздел наричаме компактен алгоритъм, което идва от самата реализация, която разглеждаме като паралелно събиране на два вектора. Тук също така използваме споделена памет, локални регистри и шаблон за пренареждане на паметта. Този подход на реализация при първоначалните изследвания дава резултати, близки до най-добрия алгоритъм, представен в подраздел 3.1.1 (Алгоритъм 3.6), но още не е направен подробен анализ и сравнение. Недостатък представлява това, че изчислява Walsh коефициентите в обратна посока. Идеята на този алгоритъм е съвместна работа с Илия Буюклиев, но до момента още не е публикуван.

Глава 4

BoolSPLG: Паралелна библиотека за изчисление на някои свойства на булеви и векторни булеви функции

Алгоритмите, основани на бътерфлай диаграма са естествено адаптирани за многопроцесорните компютърни архитектури. Някои от алгоритмите за трансформация (сродни с трансформацията на Фурие) може да се реализират с използване на бътерфлай диаграмата. В тази глава представяме паралелна реализация на библиотека за изчисление на някои свойства на булевите и векторните булеви функции, Boolean S-box Properties Library for GPUs (BoolSPLG). Библиотеката се основава на методология, която използва серия от изграждащи функции. При реализацията на библиотеката се обръща специално внимание на гъвкавостта и адаптивността. Тук ще покажем подход за проектиране на паралелни алгоритми, при които се получава оптимална производителност. Тази библиотека е необходима за нашите изследвания и освен това ще бъде достъпна до всеки, който иска да я използва. Използваната методологията ни осигурява компактност и редуцира сложността на кода без това да влияе на производителността.

Получените експериментални резултати за сравнение на последователната с паралелната реализация и полученото ускорение са представени в края на главата.

4.1 Въведение

Алгоритмите за трансформация (Fourier-related transforms) намира широко приложение в много области (свързани с дискретната математика) като крип-

тографията, теорията на кодирането, компресия на данните и т.н. Булевите функции са основни обекти в дискретната математика. Задачите, свързани с представянето, дефинирането и пресмятането на най-важните криптографски свойства и параметри на булевите и векторните булеви функции, изискват ефективни алгоритми. С увеличаване на размера на входните данни, за решаване на задачата се изисква все по-голям изчислителен ресурс. За изчисление на някои от криптографските характеристики (линейност, автокорелация, алгебрична степен, диференциална еднаквост) е необходима реализация на ефективни бътерфлай алгоритми. Както вече показахме в Глава 3, Fast Walsh (Hadamard, Walsh-Hadamard, Walsh-Fourier) Transform [P8, P5, P3] и Fast Möbius (Reed-Muller) Transform [P7] имат ефективна паралелна реализация.

Нашите изследвания се отнасят основно до конструиране на векторни булеви функции с добри криптографски свойства [P6]. Броят на получените векторни булеви функции е огромен и затова избираме само тези с добри криптографски свойства. Поради това имаме нужда от бързи алгоритми за изчисления. За нуждите на нашите изследвания ние разработваме библиотека, която съдържа паралелни алгоритми, написани на CUDA C за GPU [P9].

CUDA ориентираната библиотека ще ни позволи да изследваме криптографските свойства и параметри на големи булеви и векторни булеви функции. Библиотеката съдържа набор от различни процедури (паралелни функции), които могат да се ползват за разработка на други алгоритми, без писане на сложен код и за по-кратко време. Предназначението на библиотеката е да улесни и ускори разработването на приложения, които конструират векторни булеви функции.

Както вече споменахме (раздел 3.1) има някои инструменти, библиотеки, математически софтуер, които са свързани с изчисление на криптографските свойства на булевите и векторните булеви функции и са реализирани последователно. Тези инструменти са добри за целите на обучение и основни изчисления, но когато имаме по специфични изисквания по отношение на размера на задачата или времето на изпълнение, се нуждаем от по-специализиран софтуер или библиотеки.

До сега съществуват *GPU* библиотеки за линейна алгебра CUBLAS [44], MAGMA-ICL [41], cuSPARSE [46]. Но не са ни известни *GPU* библиотеки за изчисление на криптографските свойства на булеви и векторни булеви функции. Също така има *GPU* библиотеки, в които са реализирани бътерфлай алгоритми BPLG [38], NVIDIA's cuFFT [45], но повечето от тях са за трансформация на сигнали (Fast Fourier Transform (FFT), Hartley transform и т.н.). Те не са приложими за изследване на векторни булеви функции.

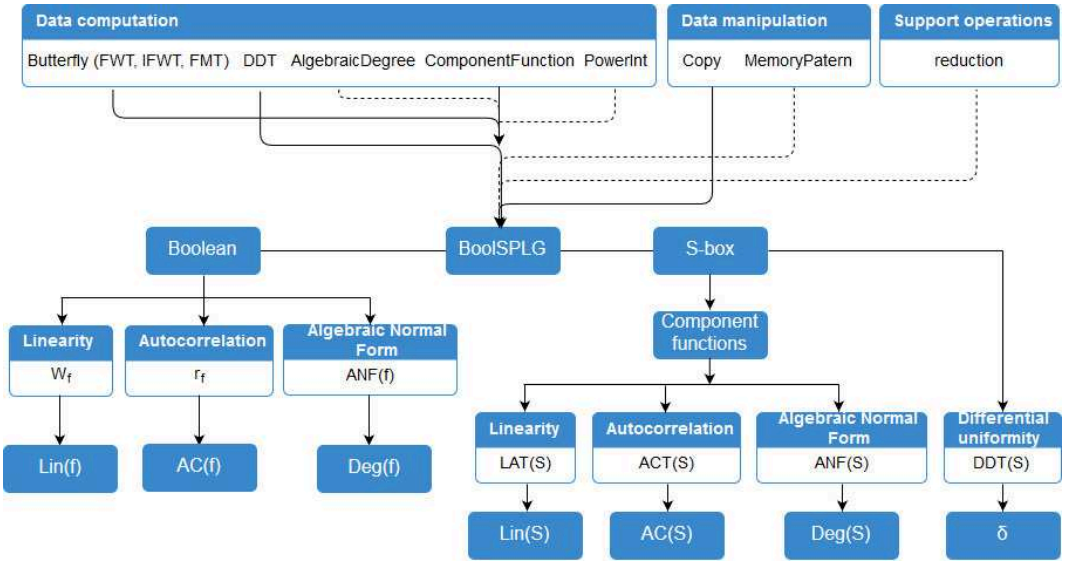
Математическите дефиниции на криптографските свойства на булевите и

векторните булеви функции са представени в Глава 1. Подробно математическо описание и алгоритми за изчисление на линейността (спектъра на Walsh) и алгебричната нормална форма са представени в Глава 2.

4.2 BoolSPLG основни процедури

В тази раздел ще опишем основните процедури, които съдържа разработената библиотека. BoolSPLG (Boolean S-box Properties Library for GPUs) е библиотека, която съдържа процедури за анализ и изчисление на криптографските свойства на булевите и векторните булеви функции. Освен процедури (паралелни функции) за оценка на криптографски характеристики са реализирани и помощни процедури.

Изграждащите блокове на алгоритмите са организирани в няколко слоя. Възможно е по-ефективно генериране на изпълним файл, като е предоставена допълнителна информация за CUDA *nvcc* компилатора (размера на входните данни, конфигурация на грида и т.н.). Нашите процедури автоматично настройват грида. Повечето от функциите са проектирани да правят изчисления на ниво регистри поради съображенията за бързодействие. Имплементираните паралелни функции комбинират използваемата памет (регистри, локална памет, споделена памет) така, че да се осигури по-висока пропускателна способност.



Фигура 4.1: Класификация и зависимост помежду изграждащите блокове на *BoolSPL* библиотеката

Предложената библиотека реализира процедури с комбиниране на основните паралелни функции в параметризирани процедури. Изграждащите функции са разделени в три групи: за изчисления на данни (Butterfly (FWT, IFWT, FMT), DDT, AlgebraicDegree, ComponentFunction, PowerInt), манипулация на данни (Copy, MemoryPatern) и помощни функции (reduction). Фигура 4.1, представя схема на класификация и зависимост между изграждащите блокове на $BoolSPLG$ библиотеката. Плътната линия показва зависимост, а пунктираната линия представлява възможен компонент. Нашата библиотека съдържа следните алгоритми: binary Fast Walsh Transforms (FWT), binary Inverse Fast Walsh Transforms (IFWT), binary Fast Möbius Transforms (FMT). Освен това има и допълнителни алгоритми (алгоритми за изчисление на DDT, алгебрична нормална форма, компонентни функции) помощни алгоритми и функции (reduction) за осъществяване на необходимите операции. Включените процедури в $BoolSPL$ библиотеката изчисляват следните криптографски свойства: $W_f(f)$, $Lin(f)$, $LAT(S)$, $Lin(S)$, $r_f(f)$, $AC(f)$, $ACT(S)$, $AC(S)$, $ANF(f)$, $ANF(S)$, $Deg(f)$, $Deg(S)$, $DDT(S)$, δ и S_b.

4.2.1 Паралелна реализация на основни изграждащи функции (алгоритми) на BoolSPLG библиотеката

В този подраздел ще представим основните паралелни алгоритми (паралелни функции за изчисление на данни и другите спомагателни функции), от който се изгражда библиотеката.

За описание на псевдокода на алгоритмите използваме същите обозначения като в подраздел 3.1.1, както и добавеното означение за XOR операция. Имената на функциите се задават съгласно съкращенията от алгоритмите, типа на използваната памет, използваните инструкции и програмистките техники, а думата *kernel* е равнозначна на паралелна функция.

В предната Глава (Глава 3, [P8, P7]) описахме паралелната реализация на бътерфлай алгоритмите FWT и FMT, чрез които изчисляваме Walsh спектър ($W_f(f), Lij(f)$) и алгебрична нормална форма ($ANF(f)$) (от таблицата на истинност и обратно) на булева функция. Първият алгоритъм, за който ще дадем кратко описание е $IFWT$. Този алгоритъм е подобен на FWT с малка модификация.

Inverse Fast Walsh Algorithm има за вход спектъра на Walsh $[W_f]$ на булева функция f (масив с дължина 2^n). За разлика от правата трансформация, в обратната сумата и разликата на съответните елементи се делят на две. Псевдокодът за изчисление на $IFWT$ (именуваме паралелната функ-

ad_kernel($ANF(f)$). AlgebraicDegree, Kernel

Input: The array ANF with 2^n entries
Output: The array ANF with 2^n entries

```
Init tID;
deg ← 0;    /*Declare local variable*/
value ← ANF[tID]; /*Local variable for every thread, taken from ANF*/
__syncthreads();

    deg ← __popc(tID) * value;

ANF[tID] ← deg;
```

ция **ifwt_kernel_shfl_xor_SM**) е подобен на паралелната функция на FWT (**fwt_kernel_shfl_xor_SM**) (подраздел 3.1.1), където съответната операция за нова стойност е заменена с:

$$\begin{aligned} value \leftarrow & (ii * (\_\_shfl\_xor(value, j) - value) \\ & + (1 - ii) * (\_\_shfl\_xor(value, j) + value))/2; \\ value \leftarrow & (value + tmpsdata[i + j])/2; \\ value \leftarrow & (-value + tmpsdata[i - j])/2; \end{aligned}$$

Функцията **AlgebraicDegree** изчислява алгебричната степен на булевата функция, като за вход има масив с дължина 2^n, който е $ANF(f)$. На изхода функцията дава всички алгебрични степени на булевата функция f. Паралелната функция използва целочислена Intrinsic функция ($\_\_popc(unsigned\,int\,x)$), която намира броя на значещите битове в 32 битово число [47]. За да намерим алгебрична степен (max или min), прилагаме добре известния reduction (max или min) алгоритъм. Псевдокодът на функцията $AlgebraicDegree$ е представен в **ad_kernel**.

Функцията (**ad_kernel**) първо записва стойностите от входния масив $ANF(f)$ в локалните регистри $value$. На следваща стъпка умножаваме теглото на индекса на нишката с $value$. Резултатът от умножението записваме в изходния масив.

За изграждане на някои от процедурите е необходима функция за степенуване (power function). Съществува вградена функция в CUDA [47], но като аргументи използва данни с плаваща запетая (floating point). По тази причина реализираме функция за степенуване ($powInt$) на положителни цели числа. За вход имаме целочислен масив V_f с дължина 2^n (основни стойности) и цяло

powInt_kernel(V_f, exp)**.** powInt, Kernel

Input: The array V_f with 2^n entries, and exponent (exp)
Output: The array V_f with 2^n entries

```
Init tID;
base ← V_f[tID];        /*Local variable for every thread, taken from V_f*/
__syncthreads();
result ← 1; /*Declare local variable*/

while (exp) do
    if(exp&1)
       result* = base;
      exp >>= 1;
      base* = base;
    end then
end while
V_f[tID] ← result;
```

число (експонент). Изходът е също масив V_f с дължина 2^n, който представлява степените на всички стойности на масива. Псевдокодът на паралелната функция $powInt$ е представен в **powInt_kernel**.

За изчисление на компонентните векторни булеви функции реализираме паралелна функция $ComponentFn$. За изучаване на някои от криптографските свойства (линейност, нелинейност, алгебрична степен, автокорелация) на векторните булеви функции трябва да вземем всички ненулеви линейни комбинации на координатните функции, дефиниращи векторната булева функция (подраздел 1.2.1). Реализирани са две подобни паралелни функции за изчисляване на компонентните функции. Първата изчислява всички компонентни функции (за $n \leq 10$), а втората изчислява компонентните функции една след друга (за $n > 10$). Това разделяне е породено от ограничения ресурс (памет, броя на нишки за блок). Псевдокодът на първата паралелна функция ($ComponentFnAll$) за изчисляване на компонентните функции е представен в **ComponentFnAll_kernel**.

Първият алгоритъм (**ComponentFnAll_kernel**) за изчисляване на компонентните функции използва два масива S_{box}, CF_{out} и цяло число като аргумент $block\_size$. Входният масив S_{box} (масив с дължина 2^n) съдържа векторната булева функция. Всеки един от елементите представлява цяло число, чието двоично представяне е стълб на матрицата на векторната булева функция. Изходният масив CF_{out} (масив с дължина $2^n \times 2^n$) съдържа всички компонен-

ComponentFnAll_kernel($S_{box}, CF_{out}, block\_size$) ComponentFnAll

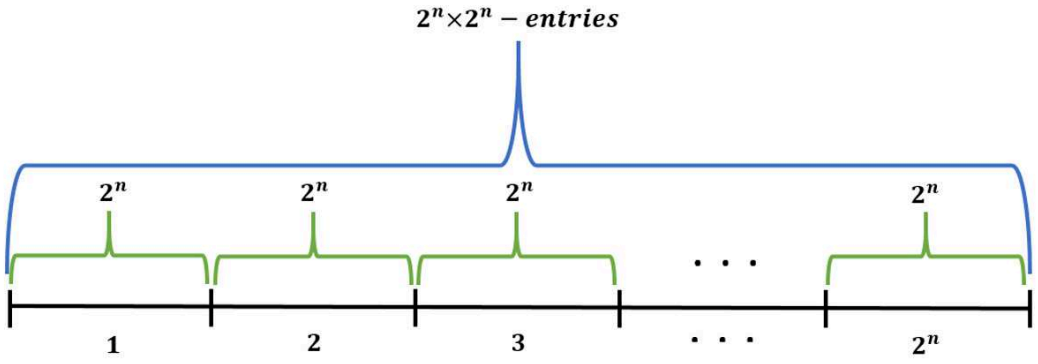
Input: The array S_{box} with 2^n entries, and $block\_size$
Output: The array CF_{out} with $2^n \times 2^n$ entries

```
Init tID, bID;
i ← bID * block_size + tID;
logI, weight, element ← 0; /*Declare local variables*/
value ← S_box[tID]; /*Local variable for every thread, taken from V_in*/
__syncthreads();

    logI ← value&bID;
    weight ← __popc(logI);
    element ← (ones&(2 − 1));   //i&(n − 1) : (i%n) => (weight%2)

CF_out[i] ← element;
```

тни функции. Броят на компонентните функции, броят на блокове, броят на нишки в блок и аргументът $block\_size$ са еднакви. Всеки блок изчислява една компонентна функция. Всеки блок има копие от векторната булева функция, всъщност стойностите от векторната булева функция се записват в локалните регистри $value$ на нишките от един блок. Изходният масив CF_{out} съдържа последователност от компонентни функции и има дължина $2^n \times 2^n$ (Фигура 4.2). Броят операции за генериране на една компонентна функция са 2^n. Поради това, че имаме 2^n компонентни функции, общият брой операции за генериране на всички компонентни функции е 2^{2n}.



Фигура 4.2: Векторна булева функция - масив съдържащ всички компонентни функции

ComponentFnVec_kernel($S_{box}, CF_{num\_out}, num\_CF$) ComponentFnVec

Input: The array S_{box} with 2^n entries, and num_CF
Output: The array CF_{num_out} with 2^n entries

```
Init tID, bID, tID_inblock;
i ← tID;
logI, weight, element ← 0; /*Declare local variables*/
value ← S_box[tID_inblock]; /*Local variable for every thread, taken from V_in*/
__syncthreads();

  logI ← value&num_CF;
  weight ← __popc(logI);
  element ← (ones&(2 − 1));   //i&(n − 1) : (i%n) => (weight%2)

CF_num_out[i] ← element;
```

Вторият алгоритъм (**ComponentFnVec_kernel**) за изчисляване на компонентните функции използва два масива S_{box}, CF_{num_out} и пореден номер на компонентната функция - целочислен аргумент num_CF. Входният масив S_{box} (масив с дължина 2^n) съдържа самата векторна булева функция, докато изходният масив CF_{num_out} (масив с дължина 2^n) съдържа поредната изчислена компонентна функция. Алгоритъмът за изчисляване на компонентни функции е подобен на първия, но тук изчисляваме само една компонентна функция. Записваме стойностите на векторната булева функция в локалните променливи *value*, като първо правим логическа операция *and* между num_CF и стойността на *value* в съответната нишка. На получения резултат намираме теглото (*weight*), като използваме Intrinsic функция $\_\_popc(unsigned\ int\ x)$. Четността на теглото определя стойността на компонентната функция. Изходният масив CF_{num_out} съдържа компонентна функция, която съответства на num_CF.

Съществуват няколко последователни алгоритъма за изчисление на таблицата $DDT(S)$ [30]. Тук представяме паралелен алгоритъм за изчисление на $DDT(S)$, който се базира на израз 1.3 от подраздел 1.2.1. Реализирани са две паралелни функции за изчисляване на $DDT(S)$. Първата изчислява цялата $DDT(S)$ таблица (за $n \leq 10$), а втората я изчислява ред по ред (за $n > 10$). Псевдокодът на първата паралелна функция ($DDTFnAll$) за изчисляване на компонентните функции е представен в **DDTFnAll_kernel**.

Първият алгоритъм (**DDTFnAll_kernel**) за изчисляване на $DDT(S)$ използва два масива S_{box}, DDT_{out} и целочислена променлива *size* като аргумент.

DDTFnAll_kernel$(S_{box}, DDT_{out}, size)$. DDTFnAll

Input: The array S_{box} with 2^n entries, and $size$
Output: The array DDT_{out} with $2^n \times 2^n$ entries

Declare shared memory as the array $tmpsdata$ of length $size$
$x_2, d_y \leftarrow 0$; /*Declare local variables*/
Init $tID\_inblock$, bID;
$tmpsdata[tID\_inblock] \leftarrow S_{box}[tID\_inblock]$;
$\_\_syncthreads()$;

$x_2 \leftarrow tID\_inblock \oplus bID$;
$d_y \leftarrow (tmpsdata[tID\_inblock] \oplus tmpsdata[x_2]) + bID \times size$;

$atomicAdd(DDT_{out}[d_y], 1)$;

DDTFnVec_kernel$(S_{box}, DDT_{out}, row)$. DDTFnVec

Input: The array S_{box} with 2^n entries, and $size$
Output: The array DDT_{out} with 2^n entries

$x_2, d_y \leftarrow 0$; /*Declare local variables*/
Init tID;

$x_2 \leftarrow tID \oplus row$;
$d_y \leftarrow (S_{box}[tID] \oplus S_{box}[x_2])$;

$atomicAdd(DDT_{out}[d_y], 1)$;

Входният масив S_{box} (масив с дължина 2^n) съдържа самата векторна булева функция, изходният масив DDT_{out} (масив с дължина $2^n \times 2^n$) представлява самата $DDT(S)$ таблица. Всеки блок изчислява един ред на $DDT(S)$ таблицата. Всеки блок има копие от векторната булева функция. Логиката на алгоритъма е представена в псевдокода. Броят операции за изчисление на един ред на $DDT(S)$ таблицата е 2^n. Поради това че имаме 2^n реда акумулираният брой операции за генериране на цялата $DDT(S)$ таблица е 2^{2n}.

Вторият алгоритъм (**DDTFnVec_kernel**) за изчисляване на $DDT(S)$ използва два масива S_{box}, DDT_{out} и аргумент row - пореден номер на реда в $DDT(S)$ таблицата. Входният масив S_{box}, (масив с дължина 2^n) съдържа самата векторна булева функция, изходния масив DDT_{out} (масив с дължина 2^n) - поредния изчислен ред на в DDT таблицата. Алгоритъмът за изчисление на

$DDT(S)$ е подобен на първия, но тук изчисляваме само един ред от $DDT(S)$ таблицата.

4.2.2 Обмен на данните между блоковете

Хардуерните ресурси (паметта, броят на нишки за блок) оказват влияние на дизайна на алгоритмите. Ако размерът на входния масив е по-голям от 2^{10} елемента, в определена стъпка от функцията е необходимо пренареждане на данните от паметта. Затова използваме шаблон за пренареждане на паметта (описан в подраздел 3.1.1, в частта за описанието на алгоритъм 3.5). Шаблонът за пренареждане на паметта използва указатели и работи автоматично със съответен брой на блокове. Той е представен в Алгоритмите 3.5, 3.6, 3.8, както и във функциите **fwt_kernel_shfl_xor_SM_MP** и **fmt_kernel_shfl_up_SM_MP** от Глава 3.

4.2.3 Операция за редукция (*reduction*)

В някои алгоритми е необходимо намиране на максимална или минимална стойност в зависимост от изследваните криптографски свойства. Широко известен паралелен алгоритъм, подходящ за задачи от този тип е така наречения алгоритъм редукция (*reduction*). Той е имплементиран като примитив в повечето съществуващи GPU CUDA библиотеки [18, 48]. Важно е да отбележим, че ние търсим абсолютната максимална или минимална стойност. По тази причина реализираме модифициран редукция алгоритъм. Тук няма да даден подробности за тази реализация, поради това че алгоритъма е широко известен.

Паралелната функция за редукция използва входен масив V_f (масив с дължина 2^n) и атрибут за размер на входния масив $size$, а на изхода получаваме максимална (max) или минимална (min) стойност. Функцията за абсолютна максимална стойност е наречена **ReductionMaxA (ReductionMaxA** $(In : V_f, size)(Out : max)$**)**, а за абсолютна минимална стойност - **ReductionMinA (ReductionMinA** $(In : V_f, size)$ $(Out : min)$**)**.

4.2.4 Конфигурация на грида

Едно от основните изисквания за получаване на по-голяма производителност е постигане на максимален паралелизъм. Това зависи от конфигурацията на грида на паралелната функция. Оптималното настройване и конфигурация на грида, е ограничено от наличните ресурси. Ресурсите, който влияят на изпълнението са броят на регистрите разпределени за нишка, споделената памет за

блок, броят на блоковете, които се изпълняват едновременно на един SM и броят на нишките в блок.

Нишките в блоковете не се изпълняват, ако нямат налични ресурси. Когато започнат изпълнение заемат ресурс, който се освобождава след приключване на работата. Важно е балансиране между броя блокове и нишки за блок за осигуряване на хардуерен ресурс, които ще позволи увеличаване на паралелизацията. Синхронизацията на ниво блок съществено се влияе от броя нишки. На практика задачата за профилиране и настройване на кода за балансиране на ресурсите отнема най-много време.

4.3 Алгоритмичен дизайн на процедурите

Процедурите представляват комбинация от функции, която обединява последователност от различни алгоритми и същевременно подържа правилното паралелно разпределение на работата.

Представените функции в предходните раздели, предоставят възможност за проектиране на различни и компактни алгоритми. Главните паралелни функции, освен комбиниране на паралелните функции, изпълняват настройване и конфигурация на грида. Всеки алгоритъм (процедура) е реализиран като главна функция, която съдържа определен брой параметри (атрибути) като: входен масив, размер на задачата и резултат.

В Глава 1 (раздел 1.1 и 1.2) накратко описахме основните криптографски свойства на булевите и векторните булеви функции, в приложение ще представим основния дизайн на някой от алгоритмите.

4.3.1 Процедури за булеви функции

Алгоритми и дефиниция за изчисление на Walsh спектъра, намиране на линейност (нелинейност) на булева функция, представихме в Глава 1, 2 и 3 (раздели 1.1, 2.3 и 3.1). Процедурата за изчисление на Walsh спектъра (намиране на линейността), представлява разширение на Алгоритъм 3.6, като е добавена операция за редукция (**ReductionMaxA**), която намира максималната абсолютна стойност на коефициентите на Walsh. Псевдокодът на процедурата за изчисление на Walsh спектъра/линейността на булева функция е представен в **WalshSpecTranBoolGPU**.

Алгоритми и дефиниция за изчисление на алгебричната нормална форма ($ANF(f)$), както и намиране на алгебричната степен ($deg(f)$) на булева функция, представихме в Глави 1, 2 и 3 (раздели 1.1, 2.3 и 3.3.1). Процедурата за

WalshSpecTranBoolGPU((In: PTT(f), n, OutCome)(Out: W_f, max))

Input: The Polarity Truth Table $PTT(f)$ of the Boolean function f, with 2^n entries
Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries, maximum absolute value max

```
Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
        size ← 2^n;
        if (size <= 1024) then
           block_size ← size;
           block_num ← 1;
        end then
        else                    /* if size > 1024 */
           block_num ← size/1024;
           block_size ← 1024;
           for_shfl ← block_num;
           if (block_num > 32) then
                 for_shfl ← 32;
           end then
        end else

W_f ← PTT(f);
fwt_kernel_shfl_SM(W_f, block_size)
if (size > 1024) then
        fwt_kernel_shfl_SM_MP(W_f, block_num, for_shfl)
end then
if (OutCome) then
   ReductionMaxA (W_f, size)
end then
Copy the result back to host
Determine output result (OutCome)
Cleanup memory
```

изчисление на алгебрична нормална форма всъщност представлява Алгоритъм 3.8. Ако искаме да намерим алгебрична степен, тогава разширяваме с паралелната функция за намиране на алгебрична степен **ad_kernel** и функцията за редукция (**ReductionMaxA**). Освен представените функции, необходима е функция за конвертиране от двоично в десетично число (преди изпълнение на алгоритъма за изчисление на алгебричната нормална форма) и обратно (след изпълнение на алгоритъма).

За да изчислим автокорелацията ($AC(f)$) (автокорелацион спектър $r_f(f)$) на булева функция f, съгласно теоремата на Wiener-Khintchine [7], вземаме квадрата на спектъра на Walsh $([W_f])^2$, над който правим $IFWT$ (представено в уравнението 1.2). Броят на операциите при изчисляване на $IFWT$ е като на FWT алгоритъма ($n2^n$). Така, за да изчислим автокорелацията $AC(f)$ на булева функция f, съгласно теоремата на Wiener-Khintchine, общият брой операции е ($n2^n + 2^n + n2^n$). Процедурата за изчисление на автокорелацията на булева функция f е подобна на **WalshSpecTranBoolGPU**, само че тук добавяме паралелната функция за изчисление на квадрати на масив **powInt_kernel**. Така получаваме $([W_f])^2$, над който прилагаме $IFWT$.

4.3.2 Процедури за векторни булеви функции

В зависимост от размера на векторната булева функция $n \leq 10$ или $10 < n < 21$ има две функции.

Процедурата за изчисление на Walsh спектъра/линейност на векторна булева функция представлява разширение на Алгоритъм 3.6, като е добавена функция за изчисление на компонентните функции **ComponentFnAll_kernel**/ **ComponentFnVec_kernel** и операция за редукция (**ReductionMaxA**), която намира максималната абсолютна стойност от коефициентите на Walsh. Псевдокодът на процедурата за изчисление на Walsh спектъра/линейността на векторна булева функция е представен в **WalshSpecTranSboxGPU**.

Тази процедура използва паралелната функция за изчисление на компонентните функции на векторната булева функция - **ComponentFnAll_kernel** или **ComponentFnVec_kernel**. Функцията за намиране на максимална абсолютна стойност, се изпълнява в зависимост от поставения параметър $OutCome$. Важно е да споменем, че като изчисляваме Walsh спектъра/линейността ($n < 11$) изчислените компонентни функции се записани в един масив с дължина $2^n \times 2^n$ (Фигура 4.2), над който изчисляваме FWT (прилагаме n стъпки), а за намиране на линейността използваме **ReductionMaxA**. Когато $10 < n < 21$ изчисляваме последователно компонентните функции и намираме Walsh спектъра.

WalshSpecTranSboxGPU((In: S_{box}, n, OutCome)(Out: W_f, max))

Input: The S-box S_{box}, with 2^n entries and $OutCome$

Output: The Walsh spectrum of S-box W_f, for $n \leq 10$, with $2^n \times 2^n$ (or for $10 < n < 21$, with 2^n) entries, max absolute value

```
Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
    size ← 2^n;
if (n ≤ 10) then
      block_size ← size;   block_num ← size;
      ComponentFnAll_kernel(S_box, W_f, block_size)
      Set Polarity of Component Functions
      fwt_kernel_shfl_SM(W_f, block_size)
      if (OutCome) then
         ReductionMaxA (W_f, size)
      end then
end then

else          /* if size > 1024 */
   ReductionArray;   /*Declare array for max value from component function*/
   block_num ← size/1024;  block_size ← 1024;  for_shfl ← block_num;
         if (block_num > 32) then
               for_shfl ← 32;
         end then
      for i ←  1 to 2^n − 1 do
         ComponentFnVec_kernel(S_box, W_f, i)
         Set Polarity of Component Function
         fwt_kernel_shfl_SM(W_f, block_size)
         fwt_kernel_shfl_SM_MP(W_f, block_num, for_shfl)
         if (OutCome) then
            ReductionArray[i] ← ReductionMaxA(W_f, size)
         end then
      end for
      if (OutCome) then
        ReductionMaxA (ReductionArray,  size)
      end then
end else
Copy the result back to host
Cleanup memory
```

Kernel	size (2^n)	CPU(ms)	P1,GPU(ms)	S_p: CPUvs.GPU
WalshSpecTranSboxGPU	2^{16}	1.52	0.033	46
ReductionMaxA	2^{16}	0.185	0.0166	15.9

Таблица 4.1: BoolSPLG: CPU срещу GPU, Platform 1

Процедурата за изчисление на алгебрична нормална форма ($ANF(S)$)/ алгебричната степен ($deg(S)$) на векторна булева функция е подобна на процедурата за изчисление на ($LAT(S)$)/ $Lin(S)$ или **WalshSpecTranSboxGPU**. Тук за изчисление на компонентните функции използваме представените паралелни функции (подраздел 4.2.1), а за намиране на максимална абсолютна стойност се използва **ReductionMaxA**. Изчислението на Walsh спектъра на векторна булева функция зависи от размера n. Тук добавяме паралелната функция за намиране на алгебрична степен **ad_kernel** и операция за редукция (**ReductionMaxA**).

Изчисляваме автокорелация $AC(S)$ (автокорелацион спектър $ACT(S)$) на векторна булева функция съгласно теоремата на Wiener-Khintchine [7]. Реализираната процедура е подобна на **WalshSpecTranSboxGPU**, като тук добавяме паралелната функция за изчисление на квадрати на масив **powInt_kernel**. Така получаваме $([W_f])^2$, над който прилагаме $IFWT$.

Процедурата за изчисление на $DDT(S)$ и δ се различава от процедурите за изчисление на другите параметри на векторна булева функция. Псевдокодът на процедурата за изчисление на $DDT(S)$ и δ е представена в **DDTSboxGPU**. Тази процедура използва представените функции за изчисление на $DDT(S)$ таблицата както и функция за намиране на максимална абсолютна стойност. Важно е да споменем, че като изчисляваме $DDT(S)$ и δ на векторна булева функция, при $n < 11$, изчислената $DDT(S)$ таблицата е записана в масив с дължина $2^n \times 2^n$ (Фигура 4.2). За целия масив използваме **ReductionMaxA** операцията която намира δ. Когато $10 < n < 21$, изчисляваме последователно редовете на $DDT(S)$ таблицата, след което търсим абсолютна максимална стойност на векторна булева функция.

4.4 Експериментални резултати BoolSPLG

Представените експериментални резултати в Глава 3 са пряко свързани с библиотеката, защото използваме същите алгоритми като основни паралелни функции. За експериментите използваме платформа 1 (Таблица 3.1). Означаваме Platform 1 с $P1$.

DDTSboxGPU((In: S_{box}, n, OutCome)(Out: $DDT(S)$, δ))

Input: The S-box S_{box}, with 2^n entries and $OutCome$

Output: The $DDT(S)$ of S-box, with $2^n \times 2^n$ entries, δ absolute value

Allocate memory for device copies and host copy
Copy the input data from the host to the device
Set a grid of blocks and threads
$size \leftarrow 2^n$;
if $(n \leq 10)$ then
    $block\_size \leftarrow size$;
    $block\_num \leftarrow size$;
    DDTFnAll_kernel$(S_{box}, DDT_{out}, size)$
    if $(OutCome)$ then
        ReductionMaxA $(DDT_{out}, size \times size)$
    end then
end then
else /* if size > 1024 */
    ReductionArray; /*Array for max value from component function*/
    $block\_num \leftarrow size/1024$;
    $block\_size \leftarrow 1024$;
        end then
    $for\ i \leftarrow\ 1\ to\ 2^n - 1\ do$
        DDTFnVec_kernel(S_{box}, DDT_{out}, i)
        if $(OutCome)$ then
            $ReductionArray[i] \leftarrow ReductionMaxA(DDT_{out}, size)$
        end then
    end for
    if $(OutCome)$ then
        ReductionMaxA $(ReductionArray,\ size)$
    end then
end else
Copy the result back to host
Cleanup memory

Сравнение на процедурата WalshSpecTranSboxGPU и операцията ReductionMaxA с последователни реализации е дадено в Таблица 4.1. Реализираме процедурата и операцията на език за програмиране, C++ като използваме среда за разработка MS Visual Studio 2010. Всички CPU примери са изпълнени на платформа 1 (Intel i3-3110M) в Active solution configuration - Release, и Active solution platform - Win32.

4.5 Коментари

Представените алгоритми и процедури в библиотеката BoolSPLG са съвместна работа с Илия Буюклиев. Някои идеи и предварителни версии на част от алгоритмите са докладвани на [D2], [D4], [D5]. Също така някои идеи и версии на алгоритъма за изчисление на спектъра на Walsh са публикувани в [P4], [P5]. Статия за паралелната библиотека BoolSPLG се подготвя за публикуване [P9].

Глава 5

Метод за конструиране на биективни векторни булеви функции и негова паралелна реализация

В тази глава представяме метод за конструиране на биективни векторни булеви функции чрез квазициклични кодове. Получените резултати са описани в края на главата. Освен това ще дадем отговор на въпроса къде и при кои случаи можем да приложим паралелни алгоритми при конструирането на биективни векторни булеви функции по представения метод. В раздел 5.5 представяме и псевдокод на разработените паралелни алгоритми.

5.1 История на задачата

Намирането на добри векторни булеви функции е обсъждано в много статии и научни разработки в продължение на десетки години, при което са провеждани много разнообразни експерименти над тях.

За представяне, определяне и пресмятане на характеристиките и параметрите на векторните булеви функции са необходими ефективни алгоритми. Нарастването на размера на входните данни води до усложняване на изчисленията, които стават по-трудни, с прекалено голям брой обекти. От друга страна, някои от алгоритмите са подходящи за паралелна реализация, която дава възможност за обхождане на междинните обекти едновременно и с това позволява изследването на векторни булеви функции със сравнително големи

размери.

Обратимите векторни булеви функции от размер $n = 4$ са широко и подробно изследвани, направена е класификация [36, 57, 64], дефинирани са критерии за оптималност [36]. Обща класификация на всички оптимални векторни булеви функции за размер $n = 4$ е дадена в работата на Leander и Poschmann [36] през 2007 г. В нея авторите правят изчерпателен анализ и намират всички класове афинно еквивалентни векторни булеви функции, за които изследват линейност, диференциална еднаквост и алгебричната степен. Saarinen [57] през 2011 г. разширява работата на Leander и Poschmann, при което прави изчерпателен анализ и намира всичките класове на линейна еквивалентност, за които от основно значение са параметрите линейността и диференциална еднаквост. В статията [64] от 2015 г. се прави нова класификация на векторните булеви функции с този размер ($n = 4$), като те се разделят на 183 различни категории, получени в зависимост от стойностите на подходящо дефинирани от авторите параметри.

Почти всичко за векторните булеви функции с размер $n = 4$ е ясно, но положението при по-големите размери е коренно различно. Специален интерес представляват векторните булеви функции с размер $n = 8$. Интересът се дължи на това, че тези криптографски примитиви са внедрени в масово използваните криптографски стандарти. Още не е ясно дали съществуват векторни булеви функции за $n = 8$ с нелинейност, по-голяма от 112 (според границата на Парсевал) или други по-добри криптографски свойства. Поради това намирането на векторни булева функция с по-добри "оптимални" криптографски свойства е много актуален проблем.

5.2 Квазициклични кодове

В този раздел представяме основните дефиниции и твърдения, свързани с квазицикличните кодове, които използваме в нашата конструкция.

Даден линеен код е квазицикличен, ако всяко циклично отместване на кодова дума с $s \geq 1$ позиции дава като резултат друга кодова дума. Ако $s = 1$, кодът е цикличен, така че квазицикличните (QC) кодове са обобщение на цикличните кодове. Конструирани са много QC кодове, които имат най-голямото минимално разстояние в класа от линейни кодове с дадени дължина и размерност.

Съществуват много методи за конструиране на добри QC кодове. Един QC код с дължина lm и индекс l се поражда от клетъчна матрица, чиито клетки са $m \times m$ циркуланти. Клетъчните редове на тази матрица имат вида

$(G_1, \ldots, G_l)$, където G_i е $m \times m$ циркуланта. Казваме, че QC кода е s-generator (s-генераторен), ако броят на редовете на клетъчната матрица е s. Метод за конструиране на 1-генераторни QC кодове е представен от van Tilborg [63], а Chen [13, 14] дава констукции на 1, 2 и 3-генераторни QC кодове. За да свържем QC кодовете с векторните булеви функции, разглеждаме различни (но еквивалентни) симплекс кодове с дължина $2^k - 1$ и размерност k като QC кодове.

Нека $K = \mathbb{F}_{q^n}$ е крайно поле, α е негов примитивен елемент, $q^n - 1 = m \cdot r$ и $\beta = \alpha^r$. Ако $G = \langle \beta \rangle < K^*$, тогава G е циклична група от ред m и $G, \alpha G, \alpha^2 G, \ldots, \alpha^{r-1} G$ са всички различни съседни класове на G в K^*. За всяко $a \in \mathbb{Z}_r$ дефинираме циркулантна $m \times m$ матрица

$$C_a = \begin{pmatrix} Tr(\alpha^a) & Tr(\alpha^a\beta) & \cdots & Tr(\alpha^a\beta^{m-1}) \\ Tr(\alpha^a\beta^{m-1}) & Tr(\alpha^a) & \cdots & Tr(\alpha^a\beta^{m-2}) \\ & & \vdots & \\ Tr(\alpha^a\beta) & Tr(\alpha^a\beta^2) & \cdots & Tr(\alpha^a) \end{pmatrix}. \tag{5.1}$$

Когато m и r са взаимно прости, матриците C_a съответстват на различните съседни класове на G в K^*.

Теорема 5.1. *Ако m и r са взаимно прости, кодът $C(0)$, чиито ненулеви кодови думи са редове на матрицата $(C_0\ C_1\ \cdots\ C_{r-1})^T$, е неразложим цикличен код с дължина m и размерност $ord_m(q)$. Кодът $C(M)$, чиито ненулеви кодови думи са редовете на матрицата*

$$M = \begin{pmatrix} C_0 & C_1 & \ldots & C_{r-1} \\ C_{r-1} & C_0 & \ldots & C_{r-2} \\ & & \vdots & \\ C_1 & C_2 & \ldots & C_0 \end{pmatrix} \tag{5.2}$$

е еквивалентен на $[2^n - 1 = mr, n, 2^{n-1}]$ симплекс кода S_n.

Теорема 5.1 е доказана в [9].

Нека $\overline{M}$ е разширена матрица на M с добавен нулев стълб в началото,

$$\overline{M} = \begin{pmatrix} 0 & \\ \vdots & M \\ 0 & \end{pmatrix}, \quad M = \begin{pmatrix} C_0 & C_1 & \ldots & C_{r-1} \\ C_{r-1} & C_0 & \ldots & C_{r-2} \\ & & \vdots & \\ C_1 & C_2 & \ldots & C_0 \end{pmatrix} \sim S_n,$$

а $C(\overline{M})$ е кодът, чиито кодови думи са редове на $\overline{M}$, за $q = 2$. Тогава всяка пораждаща матрица на $C(\overline{M})$ може да се разглежда като обратима (биективна) векторна булева функция. Такива векторни булеви функции ще наричаме QCS-boxes.

5.3 Общи положения на метода за конструкция на биективни векторни булеви функции

Както вече споменахме, всичко е известно за векторните булеви функции с размер $n = 4$. Положението при по-големи стойности на n е коренно различно. Отбелязахме, че от специален интерес са векторните булеви функции с размер $n = 8$. Но въпреки усилията на десетки различни автори, те още не са класифицирани, нито пък е ясно дали има векторни булеви функции за $n = 8$ с нелинейност, по-голяма от 112 (според границата на Парсевал), или с други по-добри криптографски свойства [28, 29].

Тъй като възможните векторни булеви функции с размер n е $(2^n)!$, намирането на оптимални векторни булеви функции за $n \geq 5$ с изчерпващо търсене е почти невъзможно с наличните изчислителни ресурси. Поради тази причина правим изчерпащо търсене за конкретен вид структура. Използваме връзката между векторните булеви функции и симплекс кода, като разглеждаме пораждащи матрици на симплекс кода в циркулантна форма. В нашата конструкция симплекс кода е представен като квазицикличен код.

5.4 Конструкции на обратими векторни булеви функции чрез квазициклични кодове

Тук ще разгледаме три конструкции на векторни булеви функции чрез квазициклични кодове (QCS-boxes).

Първа конструкция (съкратено 1C):

- Вземаме първите ml реда на матрицата $\overline{M}$ така, че получената матрица G_m има ранг n, с нулев стълб в началото.
- След това изследваме всички QCS-boxes $G_m\pi$ където $\pi \in S_r$ е пермутация на циркулантите $C_0, C_1, \ldots, C_{r-1}$.

Векторните булеви функции, които се получават чрез тази конструкция, нямат добра нелинейност. Тази конструкция е естествена, но втората и третата конструкции са по-важни, защото дават по-добри резултати.

Втора конструкция (съкратено 2C):

- Отново вземаме матрицата G_m;
- При тази конструкция разглеждаме код с пораждаща матрица:

$$MR = \left(\begin{array}{c|c} 1 & 11\ldots1 \\ \hline 0 & G_m \end{array}\right) \tag{5.3}$$

  Тази матрица поражда код, който е еквивалентен на $RM(1,n)$, но има структура на квазицикличен код;

- Отново използваме матрицата $G_m\pi$, но сега изчисляваме минималното разстояние d на кода с пораждаща матрица:

$$\left(\begin{array}{c|c} 1 & 11\ldots1 \\ \hline 0 & G_m \\ 0 & G_m\pi \end{array}\right);$$

- Ако σ е пермутация, която изобразява кода на Reed-Muller $RM(1,n)$ в кода с пораждаща матрица MR, тогава d е нелинейността на QCS-box, представен от матрицата $\sigma^{-1}(G_m\pi)$.

Цикличната структура на матриците ни осигурява по-бърз алгоритъм за изчисление на линейността.

Твърдение 5.2. *Да разгледаме матриците $A = (A_0, A_1, \ldots, A_{r-1})$ и $B = (B_0, B_1, \ldots, B_{r-1})$, където A_i и B_i се $m \times m$ циркулантни матрици, $i = 0, 1, \ldots, r-1$. Ако $a_0, a_1, \ldots, a_{m-1}$ се редовете на A, а $b_0, b_1, \ldots, b_{m-1}$ се редовете на B, тогава $d(a_i, b_j) = d(a_{i+1}, b_{j+1})$ за $0 \le i, j \le m-1$ (разглеждаме $i+1$ и $j+1$ по модул m).*

Следствие 5.3. *Първите m координатни функции на QCS-box, представен с матрицата $\sigma^{-1}(G_m\pi)$ от втората конструкция, имат едно и също Walsh разпределение.*

Втората конструкция е разширение на първата, като в допълнение се използва пермутация σ, която изобразява кода на Reed-Muller $RM(1,n)$ в кода с пораждаща матрица MR.

Трета конструкция (съкратено 3C):

- За всяка от циркуланти $C_0, C_1, \ldots, C_{r-1}$ променяме реда на стълбовете, като на първо място поставяме последния стълб, на второ предпоследния, на последно първия. Това пренареждане правим за всяка циркуланта отделно, при което получаваме циркуланти $\overline{C_0}$, $\overline{C_1}$, $\ldots$, $\overline{C_{r-1}}$, които дефинират матрицата $\overline{G_m}$;

- Използваме $\overline{G_m}\pi$, където $\pi \in S_r$ е пермутация на циркулантите $\overline{C_0}$, $\overline{C_1}$, $\ldots$, $\overline{C_{r-1}}$, но сега изчисляваме минималното разстояние d на кода с пораждаща матрица:
$$\left(\begin{array}{c|c} 1 & 11\ldots1 \\ \hline 0 & G_m \\ 0 & \overline{G_m}\pi \end{array}\right);$$

- Ако σ е пермутация, която изобразява кода на Reed-Muller $RM(1,n)$ в кода с пораждаща матрица MR, тогава d е нелинейността на векторната булева функция, представена от матрицата $\sigma^{-1}(\overline{G_m}\pi)$.

Третата конструкция е разширение на втората конструкция, като добавяме представеното пренареждане на стълбовете.

Използвайки такива конструкции, можем лесно да изчислим тегловните разпределения на разглежданите кодове. Можем да проверим всички векторни булеви функции, конструирани чрез QC кодове с параметри $n \leq 8$, $r \leq 15$ и да вземем само тези, които имат голяма нелинейност.

Възможните стойности за параметрите m и r са представени в Таблица 5.1. Да отбележим, че конструкцията ни не е приложима в случаите, когато $2^n - 1$ е просто число.

5.4.1 Получени QCS-boxes, експериментални резултати

За конструиране на биективни векторни булеви функции, използваме двоични QC кодове. Както вече отбелязахме, конструираните по този начин векторни булеви функции наричаме QCS-boxes. Използваните двоични QC кодове са конструирани по два метода.

Size:$(2^n - 1)$	m (r)
$n = 4$	3(5), 5(3)
$n = 5$	31 - "просто число"
$n = 6$	21(3), 9(7), 7(9), 3(21)
$n = 7$	127 - "просто число"
$n = 8$	3(85), 5(51), 15(17), 17(15), 51(5), 85(3)
$n = 9$	7(73), 73(7)
$n = 10$	3(341), 11(93), 31(33), 33(31), 93(11), 341(3)
$n = 11$	23(89), 89(23)
$n = 12$	3(1365), 5(819), 7(585), 9(455), 13(315), 15(273), 21(195), ...
$n = 13$	8191 - "просто число"
$n = 14$	3(5461), 43(381), 127(129), 129(127), 381(43), 5461(3)
$n = 15$	7(4681), 31(1057), 151(217), 217(151), 1057(31), 4681(7)
$n = 16$	3(21845), 5(13107), 15(4369), 17(3855), 51(1285), 85(771), ...
$n = 17$	131071 - "просто число"
$n = 18$	3(87381), 7(37449), 9(29127), 19(13797), 21(12483), 27(9709), ...
$n = 19$	524287 - "просто число"
$n = 20$	3(349525), 5(209715), 11(95325), 15(69905), 25(41943), ...

Таблица 5.1: Възможни стойности на параметрите m и r

При първият метод (съкратено 1M) използваме конструкция с помощта на циркулантни матрици от вида

$$C_a = \begin{pmatrix} Tr(\alpha^{ma}) & Tr(\alpha^{r+ma}) & \cdots & Tr(\alpha^{mr-r+ma}) \\ Tr(\alpha^{r+ma}) & Tr(\alpha^{2r+ma}) & \cdots & Tr(\alpha^{ma}) \\ & & \vdots & \\ Tr(\alpha^{mr-r+ma}) & Tr(\alpha^{ma}) & \cdots & Tr(\alpha^{mr-2r+ma}) \end{pmatrix}.$$

В тази конструкция е важно числата m и r да са взаимно прости. С помощта на тези циркуланти конструираме клетъчна циркулантна матрица от вида

$$M = \begin{pmatrix} C_0 & C_1 & \ldots & C_{r-1} \\ C_1 & C_2 & \ldots & C_0 \\ & & \vdots & \\ C_{r-1} & C_0 & \ldots & C_{r-2} \end{pmatrix}. \tag{5.4}$$

Кодът, чиито ненулеви кодови думи са редовете на тази матрица, е симплекс код с параметри $[2^n - 1 = mr, s, 2^{n-1}]$.

За втория метод (съкратено 2M) разглеждаме циркуланти от вида (5.1). В този случай разглеждаме само 1-генераторни квазициклични кодове, т.е. нужно е размерността на кода да е не по-голяма от m.

С помощта на тези два метода конструираме различни клетъчни матрици от циркуланти, които пораждат симплекс код и изследваме съответните QCS-boxes за размери $4 \leq n \leq 20$. Направихме изчерпващо търсене за всички възможни размери, когато $r = 15$, а няколко случая още се изследват за $r \geq 17$. За първата конструкция няма да представим експериментални резултати, тъй като получените векторни булеви функции имат слаби криптографски свойства. Някои от получените добри QCS-boxes (с различни размери n) чрез останалите методи за конструиране са дадени в шестнадесетичен (hex) запис в приложението, докато основните криптографски свойства са представени в таблична форма в следващите подраздели на тази глава. Тук представяме всичките получени векторни булеви функции от определен размер, които се доближават до границата на Парсевал (Теорема 1.6, Таблица 5.2), до най-малката възможна стойност на δ ($\delta \geq 2$), имат максимална алгебрична степен ($deg(S)_{max}$) и минимална възможна стойност $AC(S)$, по което сравняваме с най-добрите известни векторни булеви функции (S-boxes), получени с други техники.

За втората конструкция (за двата метода на генериране на QS кодове), и всички размери n, когато увеличаваме стойността на параметъра r (m намалява), нелинейността на получените QCS-boxes има тенденция да се приближава до границата на Парсевал 1.6. Получените резултати представяме таблично,

Границата на Парсевал	$n=4$	$n=6$	$n=8$	$n=9$	$n=10$	$n=11$	$n=12$
$nl(f) \leq 2^{n-1} - 2^{n/2-1}$	6	28	120	244.68	496	1001.37	2016

Таблица 5.2: Границата на Парсевал

QC S-box for $n=4$	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
1M, $m=5, r=3$	8	4	4	3	8	3
1M, $m=3, r=5$	8	4	4	3	8	60
2M, $m=5, r=3$	8	4	4	3	8	3
2M, $m=3, r=5$	8	4	4	3	8	28

Таблица 5.3: Обратими (4×4), QCS-boxes, 2C

като в първата колона показваме използвания метод за генериране на QC код и параметрите m и r, след което следват криптографските свойства, а в последната колона представяме броя на конструираните QCS-boxes. Допълнителното обозначение звезда $*$ в последната колона означава, че този случай още се изследва или частично е изследван.

Резултати от втората конструкция (2C)

С втората конструкция получаваме QCS-boxes, чиито параметри се доближават или са еднакви с параметрите на векторните булеви функции (S-boxes) със същия размер, но получени с използването на инверсия в крайно поле. Представяме и по-подробно описание на получените QCS-boxes в зависимост от размера.

- **Размер** $n=4$
  Дефиницията за оптимални (4×4) векторни булеви функции (S-boxes) е представена в [36]. С конструкция 2C получаваме много оптимални векторни булеви функции, които описваме в Таблица 5.3. Няколко от конструираните оптимални векторни булеви функции са представени в приложението.

- **Размер** $n=6$
  Получените обратими (6×6) QCS-boxes с най-добри криптографски свойства са представени в Таблица 5.4. Няколко от тях са подробно описани в приложението.

QCS-box for $n = 6$	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
1M, $m = 9, r = 7$	16	24	4	5	16	7
	16	24	4	5	32	7
	16	24	4	5	64	7
2M, $m = 7, r = 9$	16	24	4	4	24	1

Таблица 5.4: Обратими (6×6), QCS-boxes, 2C

S-box , $n = 8$	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
AES S-box [19]	32	112	4	7	32	/
QC, 1M, $m = 17, r = 15$	32	112	4	7	32	15
QC, 1M, $m = 15, r = 17$	32	112	4	5	48	4 *
	32	112	4	5	56	4 *

Таблица 5.5: Обратими (8×8), QCS-boxes (2C), сравнение с AES S-box

- **Размер** $n = 8$
  При изследване на случаите при (8×8) QCS-boxes, в зависимост от параметрите m и r, конструираме десетки обратими векторни булеви функции, чиито параметри се доближават или имат еднаква нелинейност $nl(S) = 112$ с най-добрия известен AES S-box, вложен в алгоритъма на стандарта за блоков шифър AES [19]. Резултатите са представени в таблица 5.5. Няколко получени (8×8) QCS-boxes са дадени в приложението.

- **Размери** $n = 10$**,** $n = 12$**,** $n = 14$**,** $n = 16$ **и** $n = 18$
  За размери $n = 10$, $n = 12$, $n = 14$, $n = 16$ и $n = 18$, получаваме QCS-boxes, които имат нелинейност $nl(S) = 2^{n-1} - 2^{n/2}$. Тази граница се достига и от векторната булева функция на AES при $n = 8$. Тези оптимални стойности на нелинейността обаче са отдалечени от границата на Парсевал, която е $nl(S) = 2^{n-1} - 2^{n/2-1}$. В Таблица 5.6 са представени криптографските свойства на част от получените QCS-boxes ($n = 10$, $n = 12$, $n = 14$, $n = 16$ и $n = 18$). Полученият (10×10) QCS-box е представен в приложението. В приложението не включваме QCS-boxes за по-големите стойности на n $(n \geq 11)$ поради големия размер на матриците.

  Както и в случая $n = 8$, криптографските параметри на част от получените QCS-boxes за $n = 10$, $n = 12$, $n = 14$, $n = 16$ и $n = 18$ са подобни на параметрите на векторни булеви функции, конструирани с използването на инверсия в крайно поле $GF(2^n)$ [52].

QCS-boxes	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
QC, 1M, $n = 10$, $m = 33, r = 31$	64	480	4	9	64	1 *
QC, 1M, $n = 12$, $m = 65, r = 63$	128	1984	4	11	128	1 *
QC, 1M, $n = 14$, $m = 129, r = 127$	256	8064	4	13	256	1 *
QC, 1M, $n = 16$, $m = 257, r = 255$	512	32512	4	15	512	1 *
QC, 1M, $n = 18$, $m = 513, r = 511$	1024	130560	4	17	1024	1 *

Таблица 5.6: Обратими ($n = 10$, $n = 12$, $n = 14$, $n = 16$ и $n = 18$), QCS-boxes, 2C

QCS-box for $n = 4$	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
1M, $m = 5, r = 3$	8	4	4	3	8	3
1M, $m = 3, r = 5$	8	4	4	3	8	60
2M, $m = 5, r = 3$	8	4	4	3	8	3
2M, $m = 3, r = 5$	8	4	4	3	8	28

Таблица 5.7: Обратими (4×4), QCS-boxes, 3C

Резултати от третата конструкция (3C)

Третата конструкция е разширение на втората конструкция, при която се добавя пренареждане на стълбовете на циркулантните матрици. По-долу представяме QCS-boxes, получени с тази конструкция.

- **Размер** $n = 4$
  Както при втората конструкция, така и тук получаваме много оптимални QCS-boxes, които представяме в Таблица 5.7. Няколко получени оптимални QCS-boxes са представени и в приложението.

- **Размер** $n = 6$

QCS-box for $n = 6$	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
1M, $m = 9, r = 7$	16	24	4	5	16	7
1M, $m = 7, r = 9$	16	24	4	5	16	18
2M, $m = 21, r = 3$	16	24	4	5	16	1
2M, $m = 9, r = 7$	16	24	4	5	16	1
2M, $m = 7, r = 9$	16	24	4	5	16	1

Таблица 5.8: Обратими (6×6), QCS-boxes, 3C

S-box, $n = 8$	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
AES S-box [19]	32	112	4	7	32	/
QC, 1M, $m = 85, r = 3$	32	112	4	7	32	3
QC, 1M, $m = 51, r = 5$	32	112	4	7	32	5
QC, 1M, $m = 17, r = 15$	32	112	4	7	32	15
QC, 1M, $m = 15, r = 17$	32	112	4	7	32	1 *
QC, 2M, $m = 85, r = 3$	32	112	4	7	32	1
QC, 2M, $m = 51, r = 5$	32	112	4	7	32	1
QC, 2M, $m = 17, r = 15$	32	112	4	7	32	1

Таблица 5.9: Обратими (8×8), QCS-boxes (3C), сравнение с AES S-box

Получените обратими (6×6) QCS-boxes с най-добри криптографски параметри са представени в Таблица 5.8. Няколко от тях са подробно описани в приложението.

- **Размер** $n = 8$
  При третата конструкция за размерност 8 получаваме по-голям брой QCS-boxes с нелинейност 112 (както е при AES) [19], и освен това получаваме добри QCS-boxes при всички изследвани случаи за параметрите m и r. Резултатите са представени в Таблица 5.9. Няколко получени (8×8) QCS-boxes са дадени в приложението.

- **Размери** $n = 10$**,** $n = 12$**,** $n = 14$**,** $n = 16$ **и** $n = 18$
  За размерите $n = 10$, $n = 12$, $n = 14$ и $n = 16$, третата конструкция за всички изследвани случаи на параметрите m и r, ни дава по-голям брой QCS-boxes с нелинейност $nl(S) = 2^{n-1} - 2^{n/2}$. В Таблица 5.10 са представени основните параметри на част от получените QCS-boxes ($n = 10$, $n = 12$, $n = 14$, $n = 16$ и $n = 18$). Конструиран (10×10) QCS-box е даден и в приложението.

  Както и в предната конструкция, криптографските свойства на част от получените QCS-boxes за $n = 10$, $n = 12$, $n = 14$ и $n = 16$ са сходни на свойствата на обратимите субституционни кутии, получени с използването на инверсия в крайното поле $GF(2^n)$ [52].

- **Размери** $n = 9$**,** $n = 11$ **и** $n = 15$

  По отношение на нечетните стойности на n, всички конструкции са приложими за $n = 9$, $n = 11$ и $n = 15$, но само при третата конструкция се

QCS-boxes	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
QC, 1M, $n = 10$, $m = 341, r = 3$	64	480	4	9	64	3
QC, 1M, $n = 10$, $m = 93, r = 11$	64	480	4	9	64	11
QC, 1M, $n = 10$, $m = 33, r = 31$	64	480	4	9	64	1 *
QC, 2M, $n = 10$, $m = 341, r = 3$	64	480	4	9	64	1
QC, 2M, $n = 10$, $m = 93, r = 11$	64	480	4	9	64	1
QC, 1M, $n = 12$, $m = 819, r = 5$	128	1984	4	11	128	5
QC, 1M, $n = 12$, $m = 585, r = 7$	128	1984	4	11	128	7
QC, 1M, $n = 12$, $m = 455, r = 9$	128	1984	4	11	128	9
QC, 1M, $n = 12$, $m = 315, r = 13$	128	1984	4	11	128	13
QC, 2M, $n = 12$, $m = 1365, r = 3$	128	1984	4	11	128	1
QC, 2M, $n = 12$, $m = 819, r = 5$	128	1984	4	11	128	1
QC, 2M, $n = 12$, $m = 585, r = 7$	128	1984	4	11	128	1
QC, 2M, $n = 12$, $m = 455, r = 9$	128	1984	4	11	128	1
QC, 2M, $n = 12$, $m = 315, r = 13$	128	1984	4	11	128	1
QC, 1M, $n = 14$, $m = 5461, r = 3$	256	8064	4	13	256	3
QC, 2M, $n = 14$, $m = 5461, r = 3$	256	8064	4	13	256	1
QC, 1M, $n = 16$, $m = 21845, r = 3$	512	32512	4	15	512	3
QC, 1M, $n = 16$, $m = 13107, r = 5$	512	32512	4	15	512	5
QC, 2M, $n = 16$, $m = 21845, r = 3$	512	32512	4	15	512	1
QC, 2M, $n = 16$, $m = 13107, r = 5$	512	32512	4	15	512	1

Таблица 5.10: Обратими ($n = 10$, $n = 12$, $n = 14$ и $n = 16$), QCS-boxes, 3C

QCS-boxes	Lin	nl	δ	$deg(S)$	$AC(S)$	брой
QC, 1M, $n = 9$, $m = 73, r = 7$	44	234	2	8	48	7
QC, 2M, $n = 9$, $m = 73, r = 7$	44	234	2	8	48	1
QC, 1M, $n = 11$, $m = 89, r = 23$	88	980	2	10	88	1 *
QC, 1M, $n = 15$, $m = 4681, r = 7$	360	16204	2	14	360	7
QC, 2M, $n = 15$, $m = 4681, r = 7$	360	16204	2	14	360	1

Таблица 5.11: Обратими ($n = 9$, $n = 11$ и $n = 15$), QCS-boxes, 3C

получават обратими векторни булеви функции, които имат нелинейност, близка до границата на Парсевал ($nl(S) = 2^{n-1} - 2^{n/2}$), имат висока алгебрична степен, ниска автокорелация и най-малката възможна стойност на δ ($\delta = 2$). В Таблица 5.11 са представени криптографските свойства на част от получените QCS-boxes ($n = 9$, $n = 11$ и $n = 15$). Конструираният (9×9) QCS-boxes е представен в приложението.

5.5 Проектиране на алгоритми за изследване на QCS-boxes

С нарастване на размера на векторната булева функция, задачата за изчисляване на различните криптографски параметри става по-трудна, с прекалено голям брой обекти. Вече показахме, че много от използваните алгоритми за пресмятане на криптографските параметри са подходящи за паралелна реализация.

Както вече описахме в предните Глави (3 и 4), реализирана е паралелна библиотека BoolSPLG [P9], която включва алгоритми за пресмятане на някои криптографските параметри и характеристики на булеви и векторни булеви функции. Процедурите в тази библиотека използваме за дизайн на алгоритми, реализиращи представените конструкции. Полза от прилагането на паралелни алгоритми имаме при големи стойности на n ($n \geq 14$, подраздел 5.5.1). Тест платформите, които използваме за експериментите, са дадени в Таблица 3.1.

Алгоритъм 5.1 представлява паралелна реализация на по-бързия алгоритъм (Твърдение 5.2), на втората конструкция. Твърдение 5.2 показва, че е достатъчно е да се генерират r на брой компонентни функции, които са достатъчни за да се пресметне линейността на целия QCS-box. Друга особеност е, че в цикъла, в който пресмятаме линейността на компонентните булеви функции, ако стигнем до функция, чиято линейност е по-голяма от предварително зададена гранична стойност, излизаме от цикъла и продължаваме с генериране на нова пермутация, която води до конструиране на нов QCS-box. В зависимост от големината на r, броят на възможните пермутации можем да разделим на части, за които определяме начална и крайна пермутация.

На входа на Алгоритъм 5.1 имаме вектора STT, размера n на конструираните QCS-boxes, пермутация σ, гранична стойност за линейността $BoundLin$, номерата на начална и крайна пермутации $startRank$ и $endRank$. Векторът STT е равен на първия ред на матрицата, която поражда съответния QC код. Чрез този вектор генерираме r на брой компонентни функции и от тях избираме n на брой линейно независими координатни функции ($r > n$).

Algorithm 5.1 QCS-boxes, втора конструкция, паралелна реализация

Input: Array STT, size n, permutation σ, $BoundLin$, $starRank$ and $endRank$
Output: The permutation π, the constructed S-box S of size n and its linearity $Lin(S)$

```
include "BoolSPLG"

// Allocate memory for device and host, copy inputs to device
// Configuration of the grid, for Kernels 1 and 2
//Set GPU constants m and r
d_STT ← STT;   /*Set device vector*/
 for rank from starRank to endRank do
 π ← GenPerm(rank);   /*Generate the permutation according to rank*/
 // Copy the permutation to the device memory (π)
 indexOrder ← −1;
  for i from 0 to r − 1 do
  SetOrderGPU(d_STT, d_Vor, indexOrder); //Kernel 1
   SetPermutationGPU(d_Vor, d_PTT, π, σ); //Kernel 2
   WalshSpecTranBoolGPU(d_PTT, n, true); //BoolSPLG procedure
    //Copy max Linearity to host
    if max > BoundLin; then   /*check the bound*/
     return;
    end then
   indexOrder + +;
  end for
 end for
//Copy result back to host
//Cleanup memory
```

На изхода алгоритъмът 5.1 дава QCS-box, чиято линейност не надвишава граничната стойност $BoundLin$, линейността $Lin(S)$ и поредната пермутацията π, която определя получения QCS-box. Главни компоненти на този алгоритъм са две паралелни функции и една процедура от представената библиотека $BoolSPLG$ в Глава 4.

Първата паралелна функция **SetOrderGPU** на първа стъпка прави циклично разместване на входния масив d_STT (първия ред на разглежданата матрица от циркуланти), което е показано на Фигура 5.1, и записва резултата в изходен масив d_Vor. Променливата $indexOrder$ определя стъпката при

SetOrderGPU$(d\_STT, d\_Vor, indexOrder)$ Kernel 1

Input: The array d_STT and the integer $indexOrder$
Output: The array d_Vor with 2^n entries

```
Init tID;
repit ← 0; step ← 0; place ← 0; placeFin ← 0;
repit ← ((m − 1) + tID)/m;
place ← ((repit + indexOrder) − ((repit + indexOrder)/r) × r);
step ← (tID − (tID/m) × m);
  if (step == 0 & tID != 0) then
    step ← m;
  end then

placeFin ← (place × m) + step;
  if (tID == 0) then
    placeFin ← 0;
  end then
d_Vor[tID] ← d_STT[placeFin];
```

SetPermutationGPU$(d\_Vor, d\_PTT, \pi, \sigma)$ Kernel 2

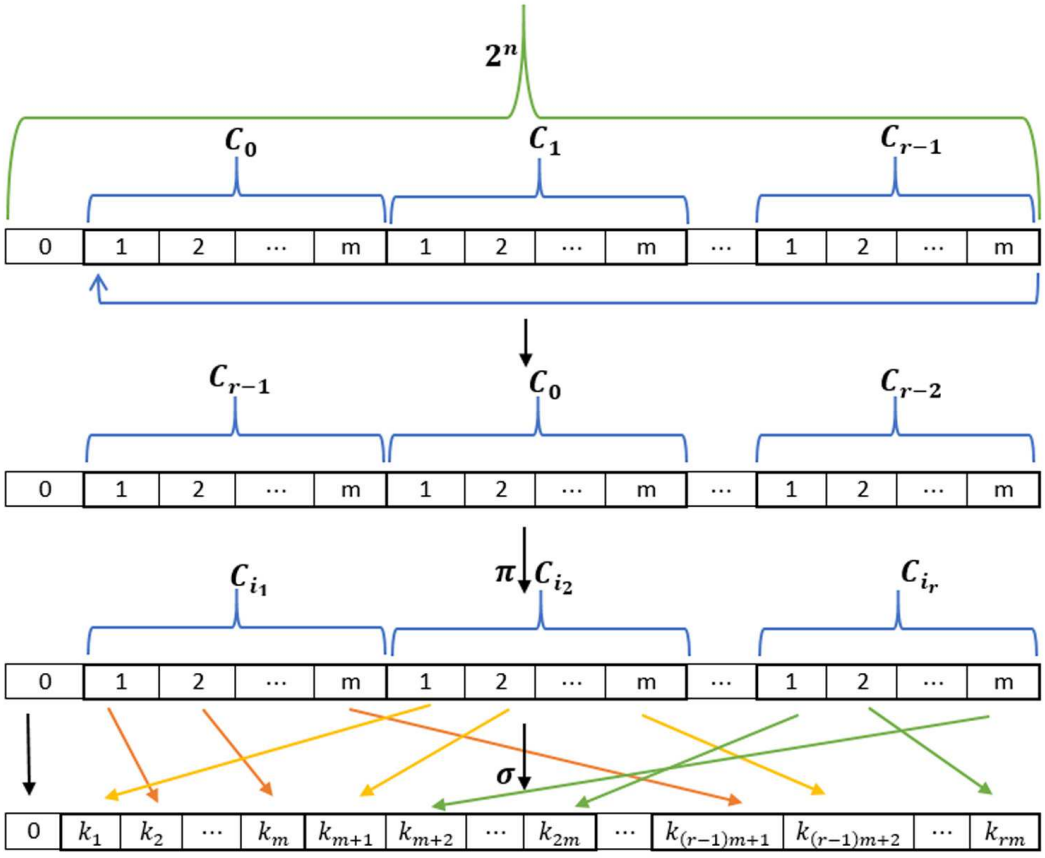
Input: The array d_Vor and the permutations π and σ
Output: The array d_PTT with 2^n entries

```
Init tID;
repit ← 0; step ← 0; place ← 0; value ← 0;
repit ← ((m − 1) + tID)/m;
step ← (tID − (tID/m) × m);
  if (step == 0 & tID != 0) then
    step ← m;
  end then
place ← (π[repit] × m) + step;
value ← d_Vor[place];
value ← 1 − (2 × value);
d_PTT[σ[tID]] ← value;
```

цикличното разместване на входния масив.

Втората паралелна функция **SetPermutationGPU** прилага пермутацията π върху входния масив d_Vor (пермутация на циркулантите), след това прилага пермутацията σ върху всички координати и накрая прави поляриза-

ция на елементите на масива (Фигура 5.1). Изходът от функцията се записва в масив d_PTT.



Фигура 5.1: Алгоритъм 5.1, втора конструкция, паралелна реализация

В следващия подраздел даваме експериментални резултати и оценка на ефективността на Алгоритъм 5.1.

5.5.1 Експериментални резултати, паралелна реализация

Тук ще представим сравнение на паралелната реализация на втората конструкция срещу последователната реализация. Псевдокод на последователната реализация на втората конструкция не е представен, но идеята на алгоритъма е подобна на паралелната реализация. Тест платформите, които използваме за експериментите, са дадени в Таблица 3.1, единствена разлика е версията на

Size	$n = 14$	$n = 16$	$n = 18$	$n = 20$
Speed up:	3 - 4	10 - 11	29 - 30	40 - 42

Таблица 5.12: Втора конструкция, GPU срещу CPU

CUDA Toolkit която е 8.0 и версията на драйверите $v378.92$. Настройките на развойната среда на платформите са вече описани в подраздел 3.1.2.

Последователният алгоритъм е реализиран с езика за програмиране C++ в MS Visual Studio 2010 развойна среда. Всички CPU примери са изпълнени на Platform 1 (Intel i3-3110M, тъй като няма значителна разлика във времето на изпълнение между платформите $\pm 5\%$) в Active solution configuration - Release, и Active solution platform - Win32.

Таблица 5.12 представя ускорението на паралелното изпълнение на втората конструкция в сравнение с последователното изпълнение. С нарастването на размера на векторната булева функция, ускорението на паралелната имплементация срещу последователната имплементация се увеличава.

5.6 Коментари

Представените конструкции са съвместна разработка с Илия Буюклиев и Стефка Буюклиева. Първите две конструкции и част от резултатите са докладвани на международната конференция *15th International Workshop on ACCT*, проведена в Албена, България, през 2016 година [D7] и публикувани в статията [P6]. Подробно описание на всичките конструкции и получени резултати се подготвят за печат съвместно с Илия Буюклиев и Стефка Буюклиева. Представеният паралелен алгоритъм е разработен съвместно с Илия Буюклиев.

Приложение

Обратими (4×4), 2C, 1M QCS-boxes

S-box No 1:
$n = 4$, $m = 5$, $r = 3$, $\pi = \{1,\ 0,\ 2\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 8$

0 b 2 7 4 5 f 3 d 9 a 1 e 8 c 6

S-box No 2:
$n = 4$, $m = 3$, $r = 5$, $\pi = \{0,\ 4,\ 2,\ 1,\ 3\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 8$

0 1 2 3 a e 6 4 f 8 7 b 5 d c 9

Обратими (4×4), 2C, 2M QCS-boxes

S-box No 1:
$n = 4$, $m = 5$, $r = 3$, $\pi = \{1,\ 0,\ 2\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 8$

0 1 7 3 d f 6 2 8 b e 9 c 4 a 5

S-box No 2:
$n = 4$, $m = 3$, $r = 5$, $\pi = \{1,\ 3,\ 4,\ 2,\ 0\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 8$

0 5 a f 1 9 4 d 2 6 e 8 3 c b 7

Обратими (6×6), 2C, 1M QCS-boxes

S-box No 1:

$n = 6$, $m = 9$, $r = 7$, $\pi = \{2,\ 1,\ 0,\ 6,\ 5,\ 4,\ 3\}$
$Lin(S) = 16$, $nl(S) = 24$, $deg(S)_{max} = 5$, $AC(S) = 16$, $\delta = 4$

0 14 29 27 13 5 e 2b 32 26 20 b 12 1d 6 17 25 11 c 4 1 23 16 f 1c 24 1a 3b 18 d 3f 2f a 33 22 15 19 10 9 3 28 2 31 7 2e 2d 2c 1f 39 2a 8 21 34 38 37 36 35 30 3c 1b 3a 3e 3d 1e

Обратими (6×6), 2C, 2M QCS-boxes

S-box No 1:
$n = 6$, $m = 7$, $r = 9$, $\pi = \{3,\ 7,\ 4,\ 0,\ 2,\ 5,\ 8,\ 1,\ 6\}$
$Lin(S) = 16$, $nl(S) = 24$, $deg(S)_{max} = 5$, $AC(S) = 16$, $\delta = 4$

0 3b 37 c 1d 26 2a 11 29 a 9 2c 3 d 30 32 1a 7 15 13 20 12 25 19 33 17 10 1f 35 4 1e 1b 14 24 22 39 21 16 5 38 3d 8 6 2d b 2b 3a 2f e 31 f 18 36 1c 34 2 27 23 2e 1 3c 28 3f 3e

Обратими (8×8), 2C, 1M QCS-boxes

S-box No 1:
$n = 8$, $m = 17$, $r = 15$, $\pi = \{5,\ 4,\ 3,\ 2,\ 1,\ 0,\ 14,\ 13,\ 12,\ 11,\ 10,\ 9,\ 8,\ 7,\ 6\}$
$Lin(S) = 32$, $nl(S) = 112$, $deg(S)_{max} = 7$, $AC(S) = 32$, $\delta = 4$

0 ef b5 de 89 6b bc d9 13 6e 51 d7 f1 79 b2 ac c8 26 dc 8 a2 a1 ec ae ff e2 f2 b4 65 ba 61 59 91 ed 41 4c a5 b9 11 2c 45 aa ca 43 27 d8 5c e6 8f fe c4 7a e4 f0 c1 68 a6 cb 74 48 c3 60 3e b3 ad 22 db 2b 82 28 bf 98 3b 4b 72 49 23 70 52 58 8a a 4e 54 53 94 87 57 4f 1f 81 b1 2e b8 cc 5f 3d 1e fd b6 88 1c 7e f4 37 c9 e1 5 83 c6 9 d1 4d bb 4a 97 3a e8 90 e3 86 d 33 c0 7c 7d 66 8c f7 5a 44 6c b7 a8 78 56 64 4 50 f6 7f da dd 30 76 20 d2 96 55 e5 93 73 47 bd f8 e0 d3 a4 b0 9f d6 15 14 df 9d 24 38 a9 85 a7 29 ab f 40 17 af 9e 5b e 3f 1b 2 63 84 5d 25 1d 71 6 99 be 46 7b 36 d4 3c 32 fb 6d ee 10 69 2a 39 5e fc e9 cf eb 6f 92 9c c2 d5 a0 b 2d 7 8d 42 12 8e 3 a3 9b 6a 19 77 34 95 2f e7 75 ce ea d0 16 21 c7 1 35 c 1a f3 67 f5 8b 80 9a f9 fa c5 cd 62 31 18

S-box No 2:
$n = 8, m = 15, r = 17, \pi = \{12,\ 6,\ 0,\ 11,\ 5,\ 16,\ 10,\ 4,\ 15,\ 9,\ 3,\ 14,\ 8,\ 2,\ 13,\ 7,\ 1\}$
$Lin(S) = 32$, $nl(S) = 112$, $deg(S)_{max} = 7$, $AC(S) = 48$, $\delta = 16$

0 42 c6 84 9c de 5a 18 29 6b ef ad b5 f7 73 31 16 a2 85 3d 5b 25 7d 17 dd 75 96 46 a b1 70 f1 2a 12 f6 80 47 f 9e d2 28 77 87 8e ed ca 6e ba 3c aa fb 54 1b 2c 8c

7a b6 5 63 4b fa f0 e3 3f 45 41 ff 34 ec 1f 1d 10 8f da 95 e 2d cd 65 a4 53 d3 15 62 f3 4e 56 4 e8 60 22 b2 c5 81 a8 68 6f df 26 a3 90 37 1a 5f 76 69 b3 88 c8 93 1 98 79 6c b 55 e6 c7 86 b8 36 e5 e0 49 19 d7 7e f5 8b 1e a5 83 ee 2b d 78 d9 4a 9a 3e 3a cb 58 30 9d 40 23 e9 94 4f fe 5d 89 7b 7 cf 33 43 6d bf a1 b9 71 2 f4 59 6a b4 e1 8a 2f ab b0 99 48 d4 b7 d1 c0 a7 3b 44 64 c4 e7 9b 13 8d bc 51 c1 8 ce fc c2 ae 4c 67 11 57 20 91 27 7f 35 3 21 be d8 9 72 52 f8 d5 50 e2 5c 38 ac bd 4d 97 cc 24 e4 a9 74 f9 a6 5e 2e 92 c 39 ea a0 14 d6 c3 66 f2 7c db c9 6 d0 82 bb dc 32 af 9f 1c fd eb 61

Обратими (10×10)**, 2C, 1M QCS-boxes**

S-box No 1:
$n = 10, m = 33, r = 31, \pi = \{10,\ 9,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0,\ 30,\ 29,\ 28,\ 27,\ 26,\ 25,\ 24,\ 23,\ 22,\ 21,\ 20,\ 19,\ 18,\ 17,\ 16,\ 15,\ 14,\ 13,\ 12,\ 11\}$
$Lin(S) = 64$, $nl(S) = 480$, $deg(S)_{max} = 9$, $AC(S) = 64$, $\delta = 4$

0 d8 1b0 36f 361 1b5 2df a0 33f 2c2 3c0 36a f9 1be 2d9 140 27f 11 184 36e 380 351 2d4 236 22a 1f2 1a 37c 225 1b3 17f 281 3f9 fe 255 23 39 309 97 2dd 301 3cc 2a3 268 1a8 3dd 6d 3ec 48 55 3e1 3e4 24b 35 100 2f8 4a 24 367 251 2ff 237 103 261 3f3 41 1fd 2a6 aa 378 46 3da 30 73 38 213 39e 12e 33a 1ba 203 38f 399 2e1 147 31 d0 141 1a4 350 1ff 3ba 72 da 273 3d9 151 91 2f7 ab 2f3 3c2 66 3c9 96 1af 6a 31a 200 2e 1f1 7a 148 95 5c 49 1ac 2cf 169 a3 1fe ae 6f 288 206 3b0 c2 316 21b 3e7 36d 83 387 3fb 120 14c 154 307 2f1 334 8c 30f 3b5 276 270 61 30d e6 133 71 68 26 33d 24a 25d 24e 274 137 375 330 7 19e 31e 77 332 a7 1c2 a5 38a 28e 44 63 35d 1a0 79 283 349 3ac 2a1 344 3ff 371 374 3cb d6 e4 135 1b4 1ec e7 f5 3b3 27c 2a2 7e 123 c4 1ee 1ef 156 1e7 293 384 cd cc cb 392 250 56 12c 22c 35f 2eb d5 b8 234 1 3df 5d 381 3e2 275 f4 3d0 291 121 12a 3a2 b9 3a3 93 bd 138 359 1bb 19f 11f 2d3 2aa 146 3fd 3a9 15d 10a de be 111 263 181 c 35e 360 34b 185 21 22d 36 16d 3cf 3e 2db 18d 106 15f 2fe 30e f3 3f7 1f8 240 194 298 2a9 f6 20e 2ce 1e3 20c 269 31c 3bd 119 3c6 21e 152 36b 32b ec 86 e1 1c1 c3 2c3 21a 193 1cc 267 329 e2 2d7 d1 3f2 4d 13d 21f 27b 333 94 215 ba 1f7 9d e8 22f 26e 3aa 2ea 12f 260 2d2 f 57 33c 165 23d 183 ef 54 70 264 2b5 14f 87 385 38b 14b 315 60 11c 221 88 1dc c7 372 158 2ba 31d 340 376 f2 1cb 107 160 292 18f 358 2e9 143 3a6 289 3fe 76 2e3 295 2e8 101 396 388 39b 1ad 1d9 1c8 2b2 26a 3c8 368 3d8 3e9 1ce 32f 1eb 34d 366 1e 3ed f8 182 145 12 fd 249 246 188 386 3dc 23f 3de 105 2ad 162 3a4 3ce 326 126 10e 308 1f5 19a 198 299 197 1fa 324 227 a1 eb ad 1df 259 3f8 58 27e 2be 1da 2c 1d6 38c 1ab 74 171 256 69 2 3a0 3be 1d bb 5 303 209 25a 3c4 4f ea 222 1e9 d 3a1 26d 122 a6 242 2b7 254 2d5 345 173 364 347 15e 127 322 17a 226 2d8 271 2c8 2b3 278 377 1a9 33e 23e 172 1a7 320 155 357 28c 1a1 3fa 362 353 114 2bb 3a5 214 c8 339 1bd 34e 17c bc 223 c5 c6 302 6e 19 3ae 2bc 196 2c0 136 10c 296 19b 30b 210 42 78 5a 6c 3b7 2da 50 39f 3e0 7c 36c 208 1b7

3a8 31b 115 20d 112 2bf 1fc 32a 21c 4b 1e6 134 3ee 3f6 224 3f0 125 80 212 328 11b 130 220 153 1bc 1ed 218 1c 1cf 19d 3c7 370 18 2a0 d2 ff 239 139 2a8 37b 379 233 d7 38d 17 3d 2a4 22e 2d6 2b4 257 144 1d8 18b 10d 3b6 1c3 290 383 39a 187 33b 338 186 99 34 325 327 9b 398 cf 3b 253 252 1c5 22 1ae 23c 3d6 1a2 1b8 3e5 26b 9a 2f6 27a 13e 3f 262 f7 149 266 265 128 2b 116 175 25c 3ef 1c0 13a 3e8 90 1d1 3d1 5e 9c dd 8f 355 1d4 285 25f 131 c0 3af 1a5 10 2b6 1f 2c6 af 37f 279 2fc 2ca 7b 167 306 38e 1de 3e3 a9 395 243 e0 161 c9 394 16b 1f9 29e 10f 199 30a 2fb 317 3d5 297 369 22b b2 c1 2a 238 35a 43 3c5 230 110 ee 3b9 2ac 18e 3bb 2e5 2b0 2c7 174 1d3 23b 14a 280 1c4 3cd 2ec 159 1e4 1f4 397 1a6 20f 1f6 2c1 9 124 3c3 31f 282 2b1 1d2 393 287 2fa 34c 2fd 113 75 2ef 3fc 13f ed 16 1c6 23a 12b 1d0 e 202 104 32d 27 311 6 336 53 35b 16a 3b2 2af 391 313 16c 164 13c d4 2b9 390 3ab 2d0 3b1 8a 3d2 64 39c 3a7 25e 62 37 3d7 2cb 29b 286 2cd 108 3c 3db 28 1f0 1b6 304 3d4 28a 89 195 25 29a 1fb 312 92 109 8d 310 2de 30c 2e7 3b8 150 7f 29c 354 3bc 6b 20b 117 15a a2 2c5 1db 348 1cd 39d 19c 24c 192 24d 21d 129 211 11e 3eb 2dc 335 17b 9f 331 a4 132 8b 32e 2e0 3f4 248 1e8 4e 247 142 98 1d7 8 15b 363 3bf 37e b3 1c7 3f1 1ca 321 b0 3ca fc 2cc 17d 1ea 3b4 59 15 3ad 1e2 318 277 356 1dd 163 e9 2a5 2e2 3e6 ac fa d3 fb 4 1e1 341 1c9 37d 17e 3a 177 29f b 11d 207 82 13 3 229 b5 157 389 b6 9e 15c 1d5 45 232 3d3 231 1b 365 343 284 14 db 3ea 244 ca 14d 189 84 16f 373 a8 14e 1aa 235 28b 51 2ed 2e6 ce 2c9 294 28f 16e 2bd 24f 52 245 170 2f4 323 4c 204 1b1 1bf e3 e5 190 1e5 166 2f5 20a f1 13b 2ee b1 352 1f3 27d 7d 2f0 2e4 bf 34f 8e 241 201 314 2ab 5b 2ae 319 118 1b2 342 a 3f5 65 2c4 1b9 2a7 11a 228 176 67 34a b7 29 2b8 191 102 df 272 2f2 37a 305 29d 258 2f9 178 5f 47 300 18a 2d 18c d9 205 32 dc 28d 33 25b 35c 81 26f 179 382 32c 2f 180 216 26c 219 346 12d 40 337 3c1 217 10b 1a3 20 1e0 85 2d1 f0 168 b4

Обратими (4×4)**, 3C, 1M QCS-boxes**

S-box No 1:
$n = 4$, $m = 5$, $r = 3$, $\pi = \{2,\ 1,\ 0\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 4$

0 3 e 1 b f 8 a 6 d 7 4 c 9 2 5

S-box No 2:
$n = 4$, $m = 3$, $r = 5$, $\pi = \{4,\ 3,\ 0,\ 1,\ 2\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 4$

0 4 c 8 a 2 9 b 5 6 1 7 f e d 3

Обратими (4×4)**, 3C, 2M QCS-boxes**

S-box No 1:
$n = 4$, $m = 5$, $r = 3$, $\pi = \{0,\ 2,\ 1\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 4$

0 9 8 4 3 c a b 2 1 6 7 5 f e d

S-box No 2:
$n = 4$, $m = 3$, $r = 5$, $\pi = \{3,\ 0,\ 4,\ 1,\ 2\}$
$Lin(S) = 8$, $nl(S) = 4$, $deg(S)_{max} = 3$, $AC(S) = 8$, $\delta = 4$

0 e 9 7 b 2 8 f 6 a 1 4 d c 5 3

Обратими (6×6)**, 3C, 1M QCS-boxes**

S-box No 1:
$n = 6$, $m = 9$, $r = 7$, $\pi = \{6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0\}$
$Lin(S) = 16$, $nl(S) = 24$, $deg(S)_{max} = 5$, $AC(S) = 16$, $\delta = 4$

0 2f 17 25 2b 1c 32 f 3a 15 21 2e 1b 19 34 7 3d 33 2a 2c 30 9 37 2 29 d 1d c 5 1a 20 23 1e a 39 1f 35 3 36 28 27 18 12 4 13 3b b 1 14 3f 6 11 e 24 26 16 3e 22 8 2d 3c 10 38 31

Обратими (6×6)**, 3C, 2M QCS-boxes**

S-box No 1:
$n = 6$, $m = 7$, $r = 9$, $\pi = \{8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0\}$
$Lin(S) = 16$, $nl(S) = 24$, $deg(S)_{max} = 5$, $AC(S) = 16$, $\delta = 4$

0 20 30 18 8 38 10 28 4 21 24 29 1e 1d 2e 35 6 2f 34 2a 17 36 3a 14 3 2c 25 1f 27 d c 1b 1 9 3b 23 a 1a 3c 1c 7 b 13 3f 19 3e e 11 2 37 3d 39 12 f 15 31 5 33 16 22 26 32 2d 2b

Обратими (8×8)**, 3C, 1M QCS-boxes**

S-box No 1:
$n = 8$, $m = 85$, $r = 3$, $\pi = \{1,\ 0,\ 2\}$
$Lin(S) = 32$, $nl(S) = 112$, $deg(S)_{max} = 7$, $AC(S) = 32$, $\delta = 4$

0 fb 56 fd 25 ab fe 32 ee 92 55 16 ff 1c a3 99 ce 77 49 9e aa 37 b 8b 7f a5 65

8e d d1 cc 61 df e7 bb 2f 24 4 d0 4f d5 e0 63 9b a9 85 c5 23 3f ba 7 d2 75 b2 c7 15 c4 86 68 f9 66 89 ed 30 ef c9 5b 73 7a dd 97 87 bc 12 82 8c e8 57 e5 27 6c ea f0 96 b1 a 2 4d d4 f3 c0 42 d3 62 91 7b b5 1f 5d 2a 83 1a 3c 69 3a 67 b0 d9 50 e3 8a fc e2 e1 f4 43 98 34 7c 11 a4 b3 44 5f 76 c8 de 18 f7 ad 4a 64 dc 2d 39 47 9c 3d 6e 17 4b ca 1b c3 be 5e 9 a0 c1 c6 53 46 74 f eb 2b 88 f2 13 da 93 b6 f5 e 78 19 ae cb d8 2c 14 5 e6 81 a6 f6 6a 54 35 79 cf 60 a1 f8 c2 e9 31 22 48 bf 90 bd 5a 94 b8 8f 38 2e 95 36 7d 41 8d a7 1e d6 10 b4 26 1d 33 5c 58 28 cd ec a8 6b 9f f1 84 45 7e 20 29 71 70 6d fa 4e ac 21 4c b9 51 9a d7 3e 8 40 52 db 9d 59 72 a2 af 80 b7 3b e4 1 6f 3 6 c

Обратими (8×8)**, 3C, 2M QCS-boxes**

S-box No 1:
$n = 8$, $m = 51$, $r = 5$, $\pi = \{4,\ 3,\ 2,\ 1,\ 0\}$
$Lin(S) = 32$, $nl(S) = 112$, $deg(S)_{max} = 7$, $AC(S) = 32$, $\delta = 4$

0 7e 2a 3f 7f 15 9f e6 a2 bf 8a d8 4f d5 e8 f3 a1 51 5f 58 c5 5 4c ec 27 9e e5 6a 48 74 79 c9 50 47 66 28 c7 af 2c 18 23 e2 2 75 26 ea 5c f6 60 93 cf 45 f2 37 64 35 a4 fa 3b 3a 8c bc e4 3 a8 fe 88 a3 84 33 94 21 1c 63 d7 a9 16 df e9 c 20 11 71 a5 81 c3 70 ba 13 76 dc f5 2e ae 7b 12 30 c4 c2 49 36 e7 22 dd f7 f9 1b b7 b2 bd b5 9a 56 52 7d cb 1d 2b 59 9d c6 1e b6 5e d3 72 1 4 fc 54 ff cc 44 b1 aa d1 42 b0 a 99 3c ca 90 92 8e cd 8f 31 46 eb d4 b9 c1 8b 6f c8 f4 77 19 6 fd 10 8 43 38 53 be d2 40 4b 86 e1 ed b8 5d 25 89 85 6c bb ef 6e 7a 6b ad 97 57 b3 3d 6d a6 9 f8 98 62 55 61 14 78 24 9b 1f 8d 73 83 91 ee 32 fb 87 a7 7c 96 d db 4a b d9 de d6 5a 67 da 4d f1 ab 29 f0 3e 1a 7 65 e 4e 2d 95 17 ac ce b4 e3 e0 34 f 9c 5b 2f 68 c0 69 39 d0 80 a0 41 82

Обратими (9×9)**, 3C, 1M QCS-boxes**

S-box No 1:
$n = 9$, $m = 73$, $r = 7$, $\pi = \{4,\ 3,\ 2,\ 1,\ 0,\ 6,\ 5\}$
$Lin(S) = 44$, $nl(S) = 234$, $deg(S)_{max} = 8$, $AC(S) = 48$, $\delta = 2$

0 12e 197 11a 1cb f6 8d 10f 105 1e5 104 7b 19b 146 1ad 87 c6 182 164 f2 176 82 6f 13d cd fc a3 5a 1d6 ec 43 103 163 fe 1c1 1dc b2 1e8 79 1a4 6b 1bb 54 141 1c9 37 1f8 19e 1bd 166 1c2 17e 101 51 123 2d eb 1e3 76 1da 121 12d 181 16b 1b1 d9 17f 1e e0 a2 1ee 11b cb 59 a1 f4 149 13c 1c5 1d2 94 35 194 dd 2a 12a be 1a0 1e4 49 1b 11e 1fc 100 1cf cc de cf b3 1aa 1e1 14b bf 1a9 9c 180 fa 28 83 191 a7 16 19a 175 157 f1 b8 13b 3a 1ed 190 26 96 b c0 1ba b5 1cc bb 1d8 15 6c 192 1bf 169 f 170 91 151 1e2 f7 6 18d b6 165 89 12c 126 50 f8 7a 132 a9 a4 1e9 9e 68 e2 9b e9 ef 4a 45 1a 99 ca 1a6 6e 115 98 195 e5 15f 1fb 1d0 139 1f2 1d7 24 62 10d 7e 8f 183 74

1fe e3 80 1ce 1e7 56 66 7c 16f 1b7 167 1f5 159 10 d5 f0 109 a5 48 5f 18 1d4 61 14e 1b4 1c0 143 17d c7 114 11f 1b9 41 1a3 1c8 1b0 53 60 10b 44 1cd 9d ba 1b6 ab ff 178 15c 9f 19d 38 1d 10a 1f6 a0 c8 73 13 17a 4b 179 5 8c 95 160 6a 1dd 19 15a 21 e6 5d 34 1ec 1f a 8 136 15b 18c c9 ed df 1f9 b4 1d9 7 1fd 1b8 1d1 148 d6 a8 193 1f1 17b 184 3 47 1c6 1b5 5b d7 1b2 3c 144 36 196 142 93 18b 129 128 55 17c 92 3d 1 199 19f 154 97 152 138 1f4 106 14f 134 af 171 75 4d 17 174 198 177 2b 125 d3 122 1c4 d 16d 113 4c 1f0 65 153 1d3 d0 137 1df 8a 133 14c 131 1ca 1f7 72 1af c5 fd 107 e8 1c7 19c ac f9 16e 1eb 20 12 90 31 c3 168 86 18e 3f 172 147 161 c1 88 13a 16c 1ff 13e 71 14 140 e7 f5 f3 118 12b d4 33 42 69 3e 11 b7 119 1db 1f3 1b3 1fa 1a2 1ac 127 108 8e 16a 1ae 78 6d 84 116 52 aa 124 2 13f 12f 70 c 15e ea 2f 130 57 1a7 189 da 27 1e0 a6 1a1 1be 67 63 1ef 18a e 18f 158 dc 40 120 187 d1 11d e4 c2 111 d8 7d 29 1ea 30 1a8 85 d2 22 32 1e6 145 4e 11c 15d db 2c 155 4 7f e1 bc 5e ae 112 4f 14d ce 1de 1c b1 81 10e 3b 185 23 fb 1d5 150 1a5 64 8b 39 58 9 1c3 bd 25 9a 1bc 162 102 77 46 1ab 14a 117 b0 186 135 c4 ee 156 2e 10c 188 ad 5c 110 b9 173

Обратими (10×10)**, 3C, 1M QCS-boxes**

S-box No 1:
$n = 10$, $m = 93$, $r = 11$, $\pi = \{8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0,\ 10,\ 9\}$
$Lin(S) = 64$, $nl(S) = 480$, $deg(S)_{max} = 9$, $AC(S) = 64$, $\delta = 4$

0 24f 23a 327 11d 219 216 393 335 28e 10c 20d 378 30b 3c9 208 19a 207 cf 147 86 128 3e9 106 2ec 1bc 185 205 1fb 3e4 304 18d 2cd 37b 1eb 103 267 a9 3c8 2a3 1a4 243 94 25c 203 3f4 83 33d 376 124 202 de c2 1b3 45 302 3d1 fd 3f2 2a7 5b 382 2c6 e7 166 5d 3b7 1bd f5 36 271 281 14f 133 54 17f 262 1e4 151 339 2d2 394 31c 121 4a 1c3 1fe 32e 3e2 101 3fa 3c 334 41 39e 91 1bb 95 2d6 292 301 194 359 6f 23e 261 2d9 2b8 383 222 181 67 1e8 114 3e8 7e 1f9 201 2f5 153 9c 22d 1c1 1f0 234 363 73 2f4 b3 1ac 142 2e 3db 1ec 14 2de 38b 27a 21b 1e6 af 338 340 1a1 a7 5f 30c 299 2a 61 158 bf 368 331 2f2 253 78 a8 39c 200 369 105 38c 3ca 18e 38a 24d 90 9d 25 2e1 137 12c 2ff 397 8c 1f1 3dd 32b 80 3fd 17e 1a8 21e 215 39a 220 33c 197 3cf 248 97 dd 29c 34 24a 36b 226 16d 349 266 180 2ca 1d0 264 3ac 237 12 11f 117 3a9 330 16c 1de 102 35c 33a 3c1 311 14a 32a c0 33 16e f4 38f 223 28a 1f4 113 298 3f 2a1 2fc 300 336 3eb 37a 2a9 295 24e 12e 173 116 e0 44 10b 2f8 161 11a 3b1 355 140 39 17a 341 13f 59 d6 23 33f a1 17 3ed 1ed 320 291 f6 a 2cc 29d 16f 175 1c5 13d e5 71 30d f3 d1 257 164 f9 19c 3a0 4 3c3 d0 2b1 53 22f 285 31 186 14c 1e1 15 136 277 230 ac 1fa f1 25f 273 1b4 198 49 2a6 379 129 ab 23c 260 287 254 1ce 112 157 100 fc 3b4 82 274 396 1c6 3e5 13 2c7 384 db 3c5 126 29a 21d 48 3f1 4e 212 77 1dc 370 29b 13e 96 1dd 3a6 37f 3cb 34f 3f8 246 325 f8 3ee 290 119 195 40 177 3fe 2e9 361 2bf d4 c6 df 10f 235 30a 3cd f2 2d3 110 19e 167 2cb 1ee fe 1e7 324 391 b5 4b c9 6e 34e 364 35a

1a 125 33e 1b5 3fb 3f3 313 2b6 1f a4 3a4 360 333 2c0 346 65 165 2e8 bc 132 286 10 1d6 11b 35d 176 209 1ff 8f 28b 326 28 3d4 398 2ba b6 2d8 1f6 2ef 81 231 1af 1ae 2d4 39d 1e0 35 131 188 2a5 2fb 395 388 337 60 19 43 3df 2b7 38d 7a 1c7 55 372 111 145 1c2 2fa 1f3 371 289 34c 2eb 13c 21f cb 350 37e 2b4 344 380 19b 2e6 1f5 20c 367 3bd 154 159 d5 34a 1b2 127 297 303 4d 2b9 8b 17b 70 23f 23b 22 85 108 179 37c 7b b0 28d 1c9 e4 1d8 1aa 2db 2a0 74 12d 1c 2bd 312 56 1a0 9f 75 32 2c 26b 1b 2f1 11 f 19f 250 3d3 1d9 b 3f6 31b 2f6 3d6 152 390 348 b8 6 27b 249 5 366 8a 3a3 14e b7 1cf ba 36e 6d e2 29e 2fe c5 272 329 238 386 3fc 3c4 79 268 122 12b 1ad 328 2b2 7c 170 307 ce 229 3d0 2 1ea 138 3e1 68 1e9 358 284 3d8 29 317 3cc 15f 342 be 218 c3 2b0 2d1 a6 f0 1 20a 319 314 9b 13b 26e 259 118 3ba 256 2fd 351 2b 278 32f 12f 139 69 4c 2da cc 3a1 c8 24 22e 353 3bc 204 275 294 255 2dd 31e 46 227 130 143 26c 3d7 12a 25d 2e7 89 217 2c3 2ab 280 282 27e 47 27f 3da 241 123 199 13a 2ea 1cb e3 1a2 2c9 1f2 9 387 163 10a 63 3c2 26d ef 3f5 1e2 e6 93 14d 156 c1 10e 224 2ae 1f8 e8 32d 27 309 1b6 134 3b 3e3 ee 3b8 27d 3bb 34d 29f 3f0 24b 120 233 2ee 1d3 2c2 18c 1bf 6b 1e5 1a7 2ce 3dc 1fc 323 16b 192 2c8 2b5 27c 3f7 3e7 3e 148 2c1 28c ca 178 10d 20 2bb 2ed 3ff 24c 50 174 1b0 3ec 62 35f 1a9 6a 263 1f7 310 26f 87 3be 31a aa 2e5 385 3e6 2e2 1d7 279 196 169 288 1cd 18 2cf 2b3 1ab 365 206 9a 2f7 7f 76 210 2f3 f7 392 1c8 1b7 e9 25a 225 ad ea 64 37 1e3 1e 3a7 3b2 236 3ad 2a4 171 d 92 115 347 39f 2dc da 1fd 18b 252 3f9 389 244 35b 251 2e0 20f 52 3d5 270 3d2 109 3b0 399 2be 17c 160 1a3 3 232 228 dc b2 374 2a2 57 25e d3 99 343 191 5c 8 eb 1ba 8d 4f d8 3ae bb 2f 187 104 8e ff 247 332 1d4 345 193 30e 214 c7 1df 3ea 1cc 2ad 182 15d 1d1 25b 36c 269 3c7 fb 377 3e0 240 66 184 318 d7 19d 3b9 2d7 190 16a 3ce 7d 183 2f0 21a 1da 98 a0 3d9 c4 352 3ef 221 37d 155 1ca 1c4 3af 32c 39b 30 356 c 135 ed 21 1ef 36f 1d2 15b 1d5 3c6 3d 6c 149 2e3 22a 28f 1b9 316 a5 88 a2 1c0 3aa e1 213 17d 2f9 7 51 1b8 144 ae 1a6 322 b9 375 9e 1b1 5e 30f 11c 265 3a8 21c 18f 3bf 15a 305 3a2 d2 38e 3c0 cd 308 33b 373 321 fa 306 3b5 141 3b3 3de 42 2aa 35e 258 2ac 26a 1db 2df 3a5 3ab d9 293 11e 22c 14b 381 354 26 e a3 15c 245 172 362 bd 239 38 31f 20b 1a5 31d 211 276 242 36a 283 84 2bc b1 3b6 1be 357 23d 58 2a8 1d 146 2e4 2c5 72 16 34b ec 2d5 107 162 36d 2af 150 3a 18a 2d 296 20e 2c4 15e 315 5a 189 22b b4 168 2d0

Заключение

За представяне, определяне и пресмятане на характеристиките на векторните булеви функции са необходими ефективни алгоритми. С нарастване на размера на векторните булеви функции (броя на променливи) изчислителната сложност на свързаните задачи нараства многократно и те стават все по-непосилни за изпълнение. Затова важна перспектива дава фактът, че алгоритмите за тяхното решаване са подходящи за паралелна реализация. Реализацията на паралелни алгоритми и паралелни изчисления през последните години е възможна и със съвременните персонални компютри, тъй като те имат няколко процесора, процесори с повече ядра или имат графични платки, които включват процесор или процесори с повече ядра.

Класификацията и намирането на векторни булеви функции с добри криптографски свойства е трудна задача, особено за размер $n \geq 8$. Нашите изследвания се отнасят основно до конструиране на векторни булеви функции с добри криптографски свойства. Тук е представен метод за конструиране на биективни векторни булеви функции чрез квазициклични кодове. За нуждите на нашите изследвания ние разработихме библиотека, която съдържа паралелни алгоритми, написани на CUDA C за GPU. CUDA ориентираната библиотека ни позволява да изследваме и изчисляваме криптографските свойства и параметри на големи булеви и векторни булеви функции.

Научни и научно–приложни приноси:

Авторът счита, че основните приноси на дисертационния труд са:

- Разработване на паралелни алгоритми за изчисление на някои криптографски параметри на булеви функции:
  1. Реализация на паралелен алгоритъм за изчисление на Walsh спектър на булева функция.
  2. Направена е сравнителна оценка на постигнатото ускорение чрез експериментални резултати на разработения алгоритъм от точка 1.
  3. Реализация на паралелен алгоритъм за пресмятане на алгебрична нормална форма на булева функция от таблицата на истинност и обратно.
  4. Направена е сравнителна оценка на постигнатото ускорение чрез експериментални резултати на разработения алгоритъм от точка 3.
- Разработена паралелна библиотека, състояща се от процедури, които изчисляват някои криптографски параметри на булеви и векторни булеви функции (паралелните алгоритми от точките 1 и 3 са част от тази библиотека):
  5. Процедурите на библиотеката свързани с булеви функции изчисляват: линейност, автокорелационен спектър, автокорелация, алгебрична нормална форма/таблица на истинност, алгебрична степен.
  6. Процедурите на библиотеката свързани с векторните булеви функции изчисляват: компонентните функции, LAT, линейност, автокорелационен спектър, алгебрична нормална форма/таблица на истинност, алгебрична степен, DDT и δ.
- Конструкции на векторни булеви функции чрез квазициклични кодове:

  7.1 Чрез втората конструкция се получават оптимални QCS-boxes за $n = 4$.

7.2 При втората конструкция за параметри $n = 8$, $m = 17$ и $r = 15$ се получават 15 QCS-boxes с криптографски свойства като най-добрия известен AES S-box [19]. За параметри $n = 8$, $m = 15$ и $r = 17$ получените QCS-boxes имат добра нелинейност (112), но другите криптографски свойства не са добри.

7.3. При втората конструкция за $n = 6$, $n = 10$, $n = 12$, $n = 14$ и $n = 16$ за определени изследвани случаи на m и r, получаваме QCS-boxes, които от криптографска гледна точка имат същите стойности като обратими ($n = 8$) векторни булеви функции по отношение на нелинейността ($nl(S) = 2^{n-1} - 2^{n/2}$).

7.4 Третата конструкция дава оптимални QCS-boxes за $n = 4$, за всички стойности на m и r.

7.5 При третата конструкция при $n = 8$ и всички изследвани стойности на параметрите m и r получаваме десетки QCS-boxes с криптографски свойства като най-добрия известен AES S-box [19].

7.6. За третата конструкция при $n = 6$, $n = 9$, $n = 10$, $n = 11$, $n = 12$, $n = 14$, $n = 15$ и $n = 16$ за всички изследвани случаи на m и r, получаваме QCS-boxes, които от криптографска гледна точка имат същите стойности като обратими ($n = 8$) векторни булеви функции по отношение на нелинейността ($nl(S) = 2^{n-1} - 2^{n/2}$).

8. Паралелна реализация на по-бързия алгоритъм, на втората конструкция.

9. Направена е сравнителна оценка на постигнатото ускорение чрез експериментални резултати на разработения алгоритъм от точка 8.

Cryptographic properties of some vector Boolean functions and parallel algorithms with CUDA

Summary

The problems related to representation, definition and computation of the most important cryptographic properties of Boolean and Vector Boolean functions require effective algorithms. With the increasing amount of input data, the problem requires more computational resources. To compute some of the cryptographic properties (linearity, autocorrelation, algebraic degree, differential uniformity) very effective (butterfly) algorithms have to be realised. These binary butterfly algorithms are suitable for parallel implementation. Modern computer architectures, the specialised large (super)computers and PC's allow parallel computing.

The developed parallel library allows us to study and compute some of the important cryptographic properties of large Boolean and Vector Boolean functions. It is difficult to construct S-boxes with good cryptographic properties for size $n \geq 8$. In this study, a construction for S-boxes using quasi-cyclic codes is presented. We obtain S-boxes with good nonlinearity.

This work is organised in five chapters, as follows:

1. The first section of **Chapter 1** contains basic definitions related to representation, definition and computation of the most important cryptographic properties of Boolean and Vector Boolean functions. We describe the used GPU general purpose parallel computing platform and the CUDA programming model in the second part. The algorithms presented in this work are suitable for parallel implementation.

2. In **Chapter 2** we describe two transformations of Boolean functions based on the binary representation of the nonnegative integers. We present corresponding sequential algorithms which are very important in cryptography. One of the most important cryptographic characteristics of the Boolean and vector Boolean functions is the algebraic degree which is related to the Algebraic

Normal Form. Here we present an algorithm for computing the Algebraic Normal Form of a Boolean function using binary Fast Möbius (Reed-Muller) Transform. Other important cryptographic characteristics of the Boolean and vector Boolean functions are nonlinearity, autocorrelation, differential uniformity. They are related to its Walsh spectrum for whose computation we use Fast Walsh Transform.

3. The purpose of **Chapter 3** is to evaluate the performance of the recent, inexpensive and widely spread NVIDIA GPUs and to use them as a tool for computing the Algebraic Normal Form (ANF) of a Boolean function from its True Table (TT) and vice versa, and also to compute the Walsh spectrum of a Boolean function. The algorithms for computing ANF and Walsh spectrum of a Boolean function in our parallel implementation use the same basic concepts as the sequential algorithms ([P2], [P7]). For the parallel adaptation, we use CUDA C and create algorithms that use various optimisation techniques and different memory types to get better performance and efficiency.

4. In **Chapter 4** we present a CUDA oriented library that allows us to study and compute some of the most important cryptographic properties of large Boolean and Vector Boolean functions. This library is based on different sets of procedures that can be used to construct other algorithms without writing a complex code and saving time for development. The most common techniques for constructing Vector Boolean functions are: algebraic constructions, pseudo-random generation and a variety of heuristic approaches. This library is designed to facilitate and accelerate the programming development of algorithms based on different methods and techniques for constructing Vector Boolean functions.

5. In **Chapter 5** we present a construction for S-boxes using quasi-cyclic codes. Based on this construction, we obtain S-boxes with good nonlinearity for different values of n. The obtained results are given at the end of the chapter. In addition, we show where and in which cases we can apply parallel algorithms in the method for constructing bijective Vector Boolean functions.

Л И Т Е Р А Т У Р А

[1] Пл. Боровска и М. Лазарова, Паралелна информационна обработка: *системни архитектури, паралелни алгоритми, паралелно програмиране*, изд. Сиела, София, 2007.

[2] К. Манев, *Увод в дискретната математика*, Нов Български Университет, София, 1996.

[3] J. A. Alvarez-Cubero and P. J. Zufiria, A C++ class for analysing vector boolean functions from a cryptographic perspective. In: Proceedings of the 2010 International Conference on Security and Cryptography (SECRYPT), 2010, pp. 512–520.

[4] G. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *In AFIPS Conference Proceedings*, Vol. 30, pages 483-485, Washington, D.C.:Thompson Book, April 1967.

[5] J. Andrade, G. Falcao, V. Silva, Optimized Fast Walsh–Hadamard Transform on GPUs for non-binary LDPC decoding. Parallel Computing, Vol. 40, 2014, No. 9, pp. 449–453.

[6] V. Bakoev and K. Manev, Fast computing of the positive polarity Reed-Muller transform over GF(2) and GF(3), Proceeding Workshop ACCT, pp. 13-21, June, 2008.

[7] K.G. Beauchamp, Applications of Walsh and related functions: With an introduction to sequence functions (Microelectronics and Signal Processing), Academic Press, London New York, 1984.

[8] E. Biham, A. Shamir: Differential cryptanalysis of des-like cryptosystems. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 2–21. Springer, Heidelberg (1990).

[9] S. Bouyuklieva, and I. Bouyukliev, *On the binary quasi-cyclic codes*, Proceedings of the Intern. Workshop OCRT, Albena, Bulgaria, (2013), 59–64.

[10] P. J. Cameron , Encyclopaedia of DesignTheory, (2004). Availaible on: http://designtheory.org/library/encyc/

[11] C. Carlet, *Boolean Functions for Cryptography and Error Correcting Codes*. In: Crama C, Hammer PL, (Eds.), *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, Cambridge University Press, 2010, pp. 257–397.

[12] C. Carlet, *Vectorial Boolean Functions for Cryptography*, in *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, Crama, Hammer, (Eds.), Cambridge University Press, 2010.

[13] E. Z. Chen, New quasi-cyclic codes from simplex codes, IEEE Trans. Inform. Theory, 53, 1193–1196, 2007.

[14] E. Z. Chen, Good quasi-cyclic codes derived from irreducible cyclic codes, Proc. of Optimal Codes and Related Topics (OC2005), Pamporovo, Bulgaria, June 17-23, 2005, 74–81.

[15] A.D. Copeland, N. B. Chang, and S. Lung, GPU accelerated decoding of high performance error correcting codes. In: Proc. 14th Annual Workshop on HPEC, Lexington, Massachusetts, USA, 2010.

[16] CUDA C Programming Guide, Availaible on: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[17] CUDA homepage, on: http://www.nvidia.com/object/cuda_home_new.html

[18] CUDA CUB Library, Availaible on: https://nvlabs.github.io/cub/

[19] J. Daeman and V. Rijmen. The design of Rijndael: AES The advanced Encryption Standard. Springer Verlag, 2002.

[20] J. Demouth, Kepler's Shuffle: Tips and Tricks. GPU Technology Conference, 2013. Availaible on: http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf

[21] M. Flynn, Very high-speed computing systems, *Proceedings of the IEEE 54(12):1901-1909*, December 1966.

[22] M. Flynn, Some computer organizations and their effectiveness, *IEEE Transactions on Computers C-21(9):948-960*, September 1972.

[23] M. Flynn and K. Rudd, Parallel architectures, *ACM Computing Surveys 28(1):67-70*, March 1996.

[24] J. J.B. Fourier, Theorie analytique de la chaleur (in French), 1822, Paris: Firmin Didot, pere et fils, OCLC 2688081.

[25] I.J. Good, The Interaction Algorithm and Practical Fourier Analysis, Journal of the Royal Statistical Society 20 (No. 2), 1958, pp. 361–372.

[26] M. Harris, Inside Pascal: NVIDIA's Newest Computing Platform, 2016. Availaible on: `https://devblogs.nvidia.com/parallelforall/inside-pascal/`

[27] M. H. Howard, *A Tutorial on Linear and Differential Cryptanalysis* . Electrical and Computer Engineering Faculty of Engineering and Applied Science Memorial University of Newfoundland.

[28] I. Hussain, T. Shah, M. A. Gondal, and W. A. Khan, *Construction of Cryptographically Strong* 8×8 *S-boxes*, World Applied Sciences Journal **13** (2011), 2389–2395.

[29] G. Ivanov, N. Nikolov, and S. Nikova, *Reversed genetic algorithms for generation of bijective S-boxes with good cryptographic properties*, Cryptography and Communications **8** (2016), 247–276.

[30] A. Joux, Algorithmic Cryptanalysis. Chapman & Hall/CRC Cryptography and Network Security Series, 2012.

[31] M. G. Karpovsky, R. S. Stankovic, J. T. Astola, Spectral Logic and Its Applications for the Design of Digital Devices. Wiley, 2008.

[32] J. Katz and Y. Lindell. Introduction to Modern Cryptography. CRC Press, Boca Raton, FL, 2007.

[33] D. B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Elsevier, 2013.

[34] J. Kurzak, D. A. Bader, J. Dongarra, Scientific Computing with Multicore and Accelerators. CRC Press, 2010.

[35] K. Lally, P. Fitzpatrick, Algebraic structure of quasi-cyclic codes, Discr. Appl. Math., 111, 157–175, 2001.

[36] G. Leander and A. Poschmann, On the Classification of 4 Bit S-Boxes, in C. Carlet and B. Sunar (Eds.): WAIFI 2007, LNCS 4547, pp. 159–176. Springer (2007).

[37] S. Ling, P. Solé, On the algebraic structure of quasi-cyclic codes I: Finite fields, IEEE Trans. Inform. Theory, 47, 2751–2760, 2001.

[38] J. Lobeiras, M. Amor, R. Doallo, BPLG: A Tuned Butterfly Processing Library for GPU Architectures. International Journal of Parallel Programming, Vol. 43, 2015, No. 6, pp. 1078–1102.

[39] P. Maciol and Krzysztof Banas, Testing Tesla architecture for scientific computing: The performance of matrix-vector product. In: Computer Science and Information Technology, IMCSIT 2008, pp. 285–291.

[40] F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, (1977).

[41] MAGMA project aims to develop a dense linear algebra library, Availaible on: `http://icl.cs.utk.edu/magma/`

[42] MATLAB platform for solving engineering and scientific problems, Availaible on: `https://www.mathworks.com/products/matlab/`

[43] M. Matsui. Linear cryptanalysis method for DES cipher. In Advances in Cryptology – EUROCRYPT'93, vol. 765 of LNCS, pp.386-397, Springer Verlag, 1994.

[44] NVIDIA: CUDA cuBLAS Library, Availaible on: `http://docs.nvidia.com/cuda/cublas/`

[45] NVIDIA: CUDA cuFFT Library, Availaible on: `http://docs.nvidia.com/cuda/cufft/`

[46] NVIDIA: CUDA cuSPARSE Library, Availaible on: `http://docs.nvidia.com/cuda/cublas/`

[47] NVIDIA: CUDA Math API, Availaible on: `http://docs.nvidia.com/cuda/cuda-math-api/`

[48] NVIDIA: C++ template library for CUDA, Availaible on: `http://docs.nvidia.com/cuda/thrust/`

[49] NVIDIA: GeForce GT 740M specification, Availaible on: `http://www.geforce.com/hardware/notebook-gpus/geforce-gt-740m`

[50] NVIDIA: GeForce GTX TITAN specification, Availaible on: `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications`

[51] NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE, 2008, p. 6. Retrieved 2014-08-07.

[52] K. Nyberg. Differentially uniform mappings for cryptography. In Advances in Cryptology EURO- CRYPT'93, volume 765 of LNCS, pages 55–64. Springer Verlag, 1994.

[53] J.D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU Computing. Proceedings of the IEEE, Vol. 96, 2008, No. 5, pp. 879–899.

[54] S. Picek, L. Batina, D. Jakobović, B. Ege, M. Golub, S-box, SET, Match: A Toolbox for S-box Analysis. In: Information Security Theory and Practice. Securing the Internet of Things, Vol. 8501 of the series Lecture Notes in Computer Science, 2014, pp. 140–149.

[55] Profiler: CUDA Toolkit Documentation, User's Guide, Availaible on: `http://docs.nvidia.com/cuda/profiler-users-guide/`

[56] M. Quinn, Parallel Programming in C with MPI and OpenMP, *McGraw Hill Higher Education, International Edition*, 2003.

[57] M. J. O. Saarinen, Cryptographic Analysis of all 4x4-bit S- boxes, in Proceedings of the 18th International Conference on Selected Areas in Cryptography, ser. SAC 11. Springer-Verlag, 2012, pp. 118–133.

[58] Sage Mathematics Software, Availaible on: `http://www.sagemath.org/`

[59] G. E. Shannon and W. Weaver. The Mathematical Theory of Communication. The University of Illinois Press, Urbana, Illinois, 1949.

[60] C. E. Shannon. Communication theory of secrecy systems. Bell System Technical Journal, 28:656–715, 1949.

[61] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS).

[62] G. E. Séguin, G. Drolet, The Trace Description of Irreducible Quasi-Cyclic Codes, IEEE Trans. Inform. Theory, 36, 1463–1466, 1990.

[63] H. C. A. van Tilborg, On quasi-cyclic codes with rate 1/m, IEEE Trans. Inform. Theory, 24, 628–630, 1978.

[64] Wentao Zhang, Zhenzhen Bao, Vincent Rijmen, Meicheng Liu, A New Classification of 4-bit Optimal S-boxes and its Application to PRESENT, RECTANGLE and SPONGENT, Cryptology ePrint Archive: Report 2015/433.

[65] Whitepaper, NVIDIA $NVLink^{TM}$ High-Speed Interconnect: Application Performance, 2014. Availaible on:
`http://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf`

[66] X. Zhang, Y. Zheng, and H. Imai, Relating differential distribution tables to other properties of substitution boxes, Designs, Codes and Cryptography, 19:45, 2000.

[67] I. I. Zhegalkin, *On the Technique of Calculating Propositions in Symbolic Logic*, Matematicheskii Sbornik, 1927, 43:9–28.

Публикации по дисертацията

[P1] D. Bikov, S. Bouyuklieva, A. Stojanova, S-boxes – parameters, characteristics and classifications, Faculty of Computer Science, UGD-Stip Yearbook, Vol 1, No. 2, (2013), pp. 47-52, ISSN: 1857- 8691

[P2] I. Bouyukliev, D, Bikov, Applications of the binary representation of integers in algorithms for boolean functions, *Proceedings of the Forty Fourth Spring Conference of the Union of Bulgarian Mathematicians SOK "Kamchia"*, (2015), pp.161-166, ISSN: 1313-3330

[P3] D. Bikov, A. Stojanova, Using GPU matrix vector multiplication for computing Walsh spectra, *Proceedings of XII International Conference ETAI*, Охрид, Македония, (2015), ISBN: 978-9989-630-76-7

[P4] D. Bikov, I. Bouyukliev, Walsh Transform Algorithm and its Parallel Implementation with CUDA on GPUs, *Proceedings of 25 YEARS FACULTY OF MATHEMATICS AND INFORMATICS*, Велико Търново, България, (2015), pp. 29-34, ISBN: 978-619-00-0419-6

[P5] D. Bikov, I. Bouyukliev, A. Stojanova, Benefit of Using Shared Memory in Implementation of Parallel FWT Algorithm with CUDA C on GPUs, *Proceedings of 7th International Conference Information Technologies and Education Development*, Zrenjanin, Serbia, (2016) pp.250-256, ISBN 978-86-7672-285-3

[P6] I. Bouyukliev, D. Bikov, S. Bouyuklieva, S-Boxes from Binary Quasi-Cyclic Codes, *Electronic Notes in Discrete Mathematics* Volume 57, (2017), pp. 67–72, SJR 0.320

[P7] Bikov D., and I. Bouyukliev, *Parallel Fast Möbius (Reed-Muller) Transform and its Implementation with CUDA on GPUs*, to appear in *Proceedings of PASCO 2017*, Kaiserslautern, Germany, (2017).

[P8] Bikov D., and I. Bouyukliev, *Parallel Fast Walsh Transform Algorithm and its Implementation with CUDA on GPUs*, Scalable Computing: Practice and Experience (2017), submitted.

[P9] Bikov D., and I. Bouyukliev, *BoolSPLG: A library with parallel algorithms for Boolean functions and S-boxes for GPU*, preprint.

Доклади по дисертацията

[D1] Д. Биков, С. Буюклиева, Булеви функции, S-Boxes, техни параметри и особености, *Ежегоден национален семинар по теория на кодирането*, Велико Търново, (2013).

[D2] Д. Биков, И. Буюклиев, Алгоритъм за пресмятане на Walsh спектър и неговата паралелна реализация на CUDA, *Национален семинар по Теория на кодирането "Стефан Додунеков"*, Велико Търново, (2014).

[D3] D. Bikov, A. Stojanova , Using GPU matrix vector multiplication for computing Walsh spectra, *XII International Conference ETAI*, Охрид, (2015).

[D4] D. Bikov, I. Bouyukliev, Walsh Transform Algorithm and its Parallel Implementation with CUDA on GPUs, *25 YEARS FACULTY OF MATHEMATICS AND INFORMATICS*, Велико Търново, (2015).

[D5] Д. Биков, И. Буюклиев, Паралелна реализация на алгоритми за пресмятане на някои параметри на булеви функции, *Семинар по "Математически основи на информатиката"*, организиран съвместно от ФМИ-ВТУ и ИМИ-БАН, Велико Търново, (2015).

[D6] Д. Биков, И. Буюклиев, Алгоритъм за пресмятане на линейност и нелинейност на S-box и неговата паралелна реализация на CUDA, *Национален семинар по Теория на кодирането "Стефан Додунеков"*, Чифлика, (2015).

[D7] D. Bikov, I. Bouyukliev, S. Bouyuklieva, S-Boxes from Binary Quasi-Cyclic Codes, *Fifteenth International Workshop on Algebraic and Combinatorial Coding Theory*, Albena, (2016).